# Efficient local search for several combinatorial optimization problems

Mirsad Buljubasic

Collège Doctoral du Languedoc-Roussillon

# THÈSE

## Pour obtenir le grade de Docteur

Délivré par

Université de Montpellier

Préparée au sein de l'école doctorale

**I2S - Information, Structures, Systèmes**

et de l'unité de recherche

**LGI2P** - Laboratoire de Génie Informatique et d'Ingénierie de Production de l'**école des mines d'Alès**

Spécialité: **Informatique**

Présentée par **Mirsad BULJUBAŠIĆ**

# Efficient local search for several combinatorial optimization problems

Soutenue le 20 novembre 2015 devant le jury composé de

| | |
|---|---|
| **M. Jin-Kao HAO, Professeur** | Rapporteur |
| *Université d'Angers* | |
| **M. Mutsunori YAGIURA, Professeur** | Rapporteur |
| *Nagoya University* | |
| **M. Miklos MOLNAR, Professeur** | Examinateur |
| *Université de Montpellier* | |
| **M. Said HANAFI, Professeur** | Examinateur |
| *Université de Valenciennes et du Hainaut-Cambrésis* | |
| **M. Michel VASQUEZ, HDR** | Directeur de thèse |
| *Ecole des Mines d'Alès* | |
| **M. Haris GAVRANOVIĆ, Professeur** | Co-encadrant de thèse |
| *International University of Sarajevo* | |

UNIVERSITÉ DE MONTPELLIER

# Abstract

This thesis focuses on the design and implementation of local search based algorithms for discrete optimization. Specifically, in this research we consider three different problems in the field of combinatorial optimization including "One-dimensional Bin Packing" (and two similar problems), "Machine Reassignment" and "Rolling Stock unit management on railway sites". The first one is a classical and well known optimization problem, while the other two are real world and very large scale problems arising in industry and have been recently proposed by Google and French Railways (SNCF) respectively. For each problem we propose a local search based heuristic algorithm and we compare our results with the best known results in the literature. Additionally, as an introduction to local search methods, two metaheuristic approaches, GRASP and Tabu Search are explained through a computational study on Set Covering Problem.

# Résumé

Cette thèse porte sur la conception et l'implémentation d'algorithmes approchés pour l'optimisation en variables discrètes. Plus particulièrement, dans cette étude nous nous intéressons à la résolution de trois problèmes combinatoires difficiles : le « Bin-Packing », la « Réaffectation de machines » et la « Gestion des rames sur les sites ferroviaires ». Le premier est un problème d'optimisation classique et bien connu, tandis que les deux autres, issus du monde industriel, ont été proposés respectivement par Google et par la SNCF. Pour chaque problème, nous proposons une approche heuristique basée sur la recherche locale et nous comparons nos résultats avec les meilleurs résultats connus dans la littérature. En outre, en guise d'introduction aux méthodes de recherche locale mise en œuvre dans cette thèse, deux métaheuristiques, GRASP et Recherche Tabou, sont présentées à travers leur application au problème de la couverture minimale.

# Contents

# List of Figures

# List of Tables

# List of Algorithms

# Introduction

Many problems in combinatorial optimization are NP-hard which implies that it is generally believed that no algorithms exist that solve each instance of such a problem to optimality using a running time that can be bounded by a polynomial in the instance size. As a consequence, much effort has been devoted to the design and analysis of algorithms that can find high quality approximative solutions in reasonable, i.e. polynomial, running times. Many of these algorithms apply some kind of neighborhood search and over the years a great variety of such local search algorithms have been proposed, applying different kinds of search strategies often inspired by optimization processes observed in nature.

Local search algorithms move from solution to solution in the space of candidate solutions (the search space) by applying local changes, until a solution deemed optimal is found or a time bound is elapsed. These algorithms are widely applied to numerous hard computational problems, including problems from computer science (particularly artificial intelligence), mathematics, operations research, engineering, and bioinformatics. For an introduction to local search techniques and their applications in combinatorial optimization, the reader is referred to the book edited by Aarts and Lenstra (Aarts and Lenstra [1997]).

This thesis focuses on the construction of effective and efficient local search based methods to tackle three combinatorial optimisation problems from different application areas. Two of these problems arise from real-world applications where essential and complex features of problems are present. The first one is Machine Reassignment problem, defined by Google, and concerns optimal assignment of processes to machines i.e. improvement of the usage of a set of machines. The

second one is Rolling Stock Problem, defined by French Railways (SNCF), which can be classified as a scheduling (planning) problem. The task is to plan train movements in terminal stations between their arrival and departure. Both problems have been proposed in ROADEF/EURO Challenge competitions in 2012 and 2014, which are international competitions jointly organized by French and European societies of Operations Research. Although these two problems come from different application domains, they share some common features: (1) they are complex due to the presence of the large set of constraints which represent the real-world restrictions, e.g. logical restrictions, resources restrictions, etc. (2) they tend to be large. Due to these two main features of the problems, in general, solving these problems is computationally challenging. To tackle these problems efficiently, in this thesis we construct solution methods based on local search. Besides those two large scale problems, we present a local search method for efficiently solving One-dimensional Bin Packing Problem (BPP), classical and well known combinatorial optimization problem. Even though the classical instances for BPP are of a significantly smaller size than those for previous two problems, solving BPP is computationally challenging as well. This is particularly due to the fact that (1) optimal solutions are usually required (in contrast to the two previously mentioned large scale problems where optimal solutions are usually not known) and (2) some of the instances are constructed in order to be "difficult". Additionally, as an introduction to local search methods, a computational study for solving Set Covering problem by GRASP and Tabu Search is presented.

Methods developed for solving Set Covering, Bin Packing and Machine Reassignment problems are pure local search approaches, meaning that local search starts from initial solutions already given (in case of Machine Reassignment) or constructed in a very simple way (First Fit heuristic for Bin Packing and simple heuristic for Set Covering). On the other hand, for Rolling Stock problem, approach combines local search with greedy heuristics and Mixed Integer Programming (MIP). Greedy heuristic (rather complex) and Integer Programming have been used in order to obtain initial feasible solutions to the problem, which are then the subject of an improvement procedures based on local search. MIP has been used in order to produce better initial solutions (combined with greedy pro-

cedure). Nevertheless, high quality solutions can be produced even when omitting the MIP part, as was the case in our solution submitted to the ROADEF/EURO Challenge 2014 competition. Proposed local search approaches for Bin Packing and SNCF Rolling Stock problems are applied on partial configurations (solutions). This local search on a partial configuration is called the Consistent Neighborhood Search (CNS) and has been proven efficient in several combinatorial optimization problems (Habet and Vasquez [2004]; Vasquez et al. [2003]). CNS has been introduced in Chapter 1 as an improvement procedure in GRASP method for solving Set Covering Problem.

Our goal was to develop effective local search algorithms for the considered problems, which are capable of obtaining high quality results in a reasonable (often very short) computation time. Since complexity of the local search methods is mainly influenced by the complexity (size) of the local search neighborhoods, we tried to use simple neighborhoods when exploring the search space. Additional elements had to be included in the strategies in order to make them effective. The most important among those possible additional features are the intensification of the search, applied in those areas of the search space that seem to be particularly appealing, and the diversification, in order to escape from poor local minima and move towards more promising areas. Key algorithm features leading to high quality results will be explained, in a corresponding chapter, for each of the considered problems.

This thesis is organized into four main chapters:

1. **GRASP and Tabu Search: Application to Set Covering Problem** This chapter presents the principles behind the GRASP (Greedy Randomized Adaptive Search Procedure) method and details its implementation in the aim of resolving large-sized instances associated with a hard combinatorial problem. The advantage of enhancing the improvement phase has also been demonstrated by adding, to the general GRASP method loop, a Tabu search on an elementary neighborhood.

2. **One-dimensional Bin Packing** Consistent neighborhood search approach

to solving the one-dimensional bin packing problem (BPP) has been presented. This local search is performed on partial and consistent solutions. Promising results have been obtained for a very wide range of benchmark instances; best known or improved solutions obtained by heuristic methods have been found for all considered instances for BPP. This method is also tested on vector packing problem (VPP) and evaluated on classical benchmarks for two-dimensional VPP (2-DVPP), in all instances yielding optimal or best-known solutions, as well as for Bin Packing Problem with Cardinality constraints.

3. **Machine Reassignment Problem** The Google research team formalized and proposed the Google Machine Reassignment problem as a subject of ROADEF/EURO Challenge 2012. The aim of the problem is to improve the usage of a set of machines. Initially, each process is assigned to a machine. In order to improve machine usage, processes can be moved from one machine to another. Possible moves are limited by constraints that address the compliance and the efficiency of improvements and assure the quality of service. The problem shares some similarities with Vector Bin Packing Problem and Generalized Assignment Problem.
   We proposes a Noisy Local Search method (NLS) for solving Machine Reassignment problem. The method, in a round-robin manner, applies the set of predefined local moves to improve the solutions along with multiple starts and noising strategy to escape the local optima. The numerical evaluations demonstrate the remarkable performance of the proposed method on MRP instances (30 official instances divided in datasets A, B and X) with up to 50,000 processes.

4. **SNCF Rolling Stock Problem** We propose a two phase approach combining mathematical programming, greedy heuristics and local search for the problem proposed in ROADEF/EURO challenge 2014, dedicated to the rolling stock management on railway sites and defined by French Railways (SNCF). The problem is extremely hard for several reasons. Most of induced sub–problems are hard problems such as assignment problem, scheduling problem, conflicts problem on track groups, platform assignment problem,

etc. In the first phase, a train assignment problem is solved with a combination of a greedy heuristic and mixed integer programming (MIP). The objective is to maximize the number of assigned departures while respecting technical constraints. The second phase consists of scheduling the trains in the station's infrastructure while minimizing number of cancelled (uncovered) departures, using a constructive heuristic. Finally, an iterative procedure based on local search is used to improve obtained results, yielding significant improvements.

# Chapter 1

# GRASP and Tabu Search: Application to Set Covering Problem

This chapter will present the principles behind the GRASP (Greedy Randomized Adaptive Search Procedure) method and offer a sample application to the Set Covering problem. The advantage of enhancing the improvement phase has also been demonstrated by adding, to the general GRASP method loop, a Tabu search on an elementary neighborhood.

Resolution of the set covering problem by GRASP mathod presented here has been inspired by the work of Feo and Resende [1995]. The method presented in Feo and Resende [1995] has been modified by adding a tabu search procedure to the general GRASP method loop. This tabu search procedure that works with partial solution (partial cover) is referred as Consistent Neighborhood Search (CNS) and has been proven efficient in several combinatorial optimization problems (Habet and Vasquez [2004]; Vasquez et al. [2003]). The search is performed on an elementary neighborhood and makes use of an exact tabu management.

Most of the method features can be found in the literature (in the same or similar form) and, therefore, we do not claim the originality of the work presented herein; this chapter serves mainly as an introduction to GRASP and Tabu search metaheuristics and Consistent Neighborhood Search procedure. Also, application

to Set Covering problem showed to be suitable due to the simplicity of the method proposed and results obtained on difficult Set Covering instances.

## 1.1 Introduction

The GRASP (Greedy Randomized Adaptive Search Procedure) method generates several configurations within the search space of a given problem, based on which it carries out an improvement phase. Relatively straightforward to implement, this method has been applied to a wide array of hard combinatorial optimization problems, including: scheduling Binato et al. [2001], quadratic assignment Pitsoulis et al. [2001], the traveling salesman Marinakis et al. [2005], and maintenance workforce scheduling Hashimoto et al. [2011]. One of the first academic papers on the GRASP method is given in Feo and Resende [1995]. The principles behind this method are clearly described and illustrated by two distinct implementation cases: one that inspired the resolution in this chapter of the minimum coverage problem, the other applied to solve the maximum independent set problem in a graph. The interested reader is referred to the annotated bibliography by P. Festa and M.G.C. Resende Festa and Resende [2002], who have presented nearly 200 references on the topic.
Moreover, the results output by this method are of similar quality to those determined using other heuristic approaches like simulated annealing, Tabu search and population algorithms.

## 1.2 General principle behind the method

The GRASP method consists of repeating a constructive phase followed by an improvement phase, provided the stop condition has not yet been met (in most instances, this condition corresponds to a computation time limit expressed, for example, in terms of number of iterations or seconds). Algorithm 1.1 below describes the generic code associated with this procedure.

The constructive phase corresponds to a greedy algorithm, during which the

**Algorithm 1.1:** GRASP procedure

```
   input : α, random seed, time limit.
1  repeat
2  |  X ← Randomised Greedy(α);
3  |  X ← Local Search(X, 𝒩);
4  |  if z(X) better than z(X*) then
5  |  |  X* ← X;
6  until CPU time > time limit;
```

step of assigning the current variable - and its value - is slightly modified so as to generate several choices rather than just a single one at each iteration. These potential choices constitute a restricted candidate list (or $RCL$), from which a candidate will be chosen at random. Once the (variable, value) pair has been established, the RCL list is updated by taking into account the current partial configuration. This step is then iterated until obtaining a complete configuration. The value associated with the particular (variable, value) pairs (as formalized by the heuristic function $\mathcal{H}$), for still unassigned variables, reflects the changes introduced by selecting previous elements. Algorithm 1.2 summarizes this configuration construction phase, which will then be improved by a local search (simple descent, tabu search or any other local modification-type heuristic). The improvement phase is determined by the neighborhood $\mathcal{N}$ implemented in an attempt to refine the solution generated by the greedy algorithm.

**Algorithm 1.2:** Randomized Greedy

```
   input: α , random seed.
1  X = {∅};
2  repeat
3  |  Assemble the RCL on the basis of heuristic ℋ and α;
4  |  Randomly select an element x_h from the RCL;
5  |  X = X ∪ {x_h};
6  |  Update ℋ;
7  until configuration X has been completed;
```

The evaluation of heuristic function $\mathcal{H}$ serves to determine the insertion of (variable, value) pairs onto the $RCL$ (*restricted candidate list*). The way in which this criterion is taken into account exerts considerable influence on the behavior exhibited during the constructive phase: if only the best (variable, value) pair is

selected relative to $\mathcal{H}$, then the same solution will often be obtained, and iterating the procedure will be of rather limited utility. If, on the other hand, all possible candidates were to be selected, the random algorithm derived would be capable of producing quite varied configurations, yet of only mediocre quality: the likelihood of the improvement phase being sufficient to yield good solutions would thus be remote. The size of the $RCL$ therefore is a determinant parameter of this method. From a pragmatic standpoint, it is simpler to manage a qualitative acceptance threshold (i.e. $\mathcal{H}(x_j)$ *better than* $\alpha \times \mathcal{H}^*$*, where* $\mathcal{H}^*$ *is the best benefit possible and* $\alpha$ *is a coefficient lying between* 0 *and* 1) for the random drawing of a new (variable, value) pair to be assigned rather than implement a list of $k$ potential candidates, which would imply a data sort or use of more complicated data structures. The terms used herein are *threshold-based* $RCL$ in the case of an acceptance threshold and *cardinality-based* $RCL$ in all other cases.
The following sections will discuss in greater detail the various GRASP method components through an application to set covering problem.

## 1.3 Set Covering Problem

Given a matrix (*with m rows and n columns*) composed solely of $0's$ and $1's$, the objective is to identify the minimum number of columns such that each row contains at least one 1 in the identified columns. One type of minimum set covering problem can be depicted by setting up an incidence matrix with the column and row entries shown below (Figure 1.1).

More generally speaking, an $n$-dimensional *cost* vector is to be considered, containing strictly positive values. The objective then consists of minimizing the total costs of columns capable of covering all rows: this minimization is known as the *Set Covering Problem*, as exemplified by the following linear formulation:

$$cover = \begin{pmatrix} 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}$$

Figure 1.1: Incidence matrix for a minimum coverage problem

$$\begin{aligned} & \texttt{min} \quad z = \sum_{j=1}^{n} cost_j \times x_j \\ \forall i \in [1, m] \quad & \sum_{j=1}^{n} cover_{ij} \times x_j \geq 1, \\ \forall j \in [1, n] \quad & x_j \in \{0, 1\}. \end{aligned} \tag{1.1}$$

For $1 \leq j \leq n$, the decision variable $x_j$ equals 1 if column $j$ is selected, and 0 otherwise. In the case of Figure 1.1 for example, $x =< 101110100 >$ constitutes a solution whose objective value $z$ is equal to 5.

If $cost_j$ equals 1 for each $j$, then the problem becomes qualified as a *Unicost Set Covering Problem*, as stated at the beginning of this section. Both the *Unicost Set Covering Problem* and more general *Set Covering Problem* are classified as combinatorial NP-hard problems Garey and Johnson [1979]; moreover, once such problems reach a certain size, their resolution within a reasonable amount of time becomes impossible by means of exact approaches. This observation justifies the implementation of heuristics approaches, like the GRASP method, to handle these instances of hard problems.

## 1.4 An initial algorithm

This section will revisit the same algorithm proposed by T. Feo and M.G.C. Resende in one of their first references on the topic Feo and Resende [1995], where the GRASP method is applied to the Unicost Set Covering problem. It will then be shown how to improve results and extend the study to the more general Set Covering problem through combining GRASP with the *Tabu* search metaheuristic.

### 1.4.1 Constructive phase

Let $x$ be the characteristic vector of all columns $X$ (whereby $x_j = 1$ if column $j$ belongs to $X$ and $x_j = 0$ otherwise): $x$ is the binary vector of the mathematical model in Figure 1.1. The objective of the greedy algorithm is to produce a configuration $x$ with $n$ binary components, whose corresponding set $X$ of columns covers all the rows. Upon each iteration (out of a total $n$), the choice of column $j$ to be added to $X$ ($x_j = 1$) will depend on the number of still uncovered rows that this column covers. As an example, the set of columns $X = \{0, 2, 3, 4, 6\}$ corresponds to the vector $x =< 101110100 >$, which is the solution to the small instance shown in Figure 1.1.
For a given column $j$, we hereby define the heuristic function $\mathcal{H}(j)$ as follows:

$$\mathcal{H}(j) = \begin{cases} \frac{\mathcal{C}(X \cup \{j\}) - \mathcal{C}(X)}{cost_j} & \text{if } x_j = 0 \\ \frac{\mathcal{C}(X \setminus \{j\}) - \mathcal{C}(X)}{cost_j} & \text{if } x_j = 1 \end{cases}$$

where $\mathcal{C}(X)$ is the number of rows covered by the set of columns $X$. The list of $RCL$ candidates is managed implicitly: $\mathcal{H}^* = \mathcal{H}(j)$ maximum is first calculated for all columns $j$ such that $x_j = 0$. The next step calls for randomly choosing a column $h$ such that $x_h = 0$ and $\mathcal{H}(h) \geq \alpha \times \mathcal{H}^*$. The pseudo-code of the *randomized* greedy algorithm is presented below in Algorithm 1.3.

The heuristic function $\mathcal{H}()$, which determines the insertion of columns into the $RCL$, is to be reevaluated at each step so as to take into account only the uncovered rows. This is the property that gives rise to the adaptive nature of the GRASP method.

Let's now consider the instance at $n = 45$ columns and $m = 330$ rows that

**Algorithm 1.3:** `greedy`$(\alpha)$

**input** : Coefficient $\alpha \in [0, 1]$
**output**: feasible vector $x$, characteristic of the set $X$ of selected columns

```
1  X = {∅};
2  repeat
3      j* ← column, such that H(j*) is maximized;
4      threshold ← α × H(j*);
5      r ← rand() modulo n;
6      for j ∈ {r, r + 1, ..., n − 1, 0, 1, ..., r − 1} do
7          if H(j) ≥ threshold then
8              break;
9      X = X ∪ {j} (add column j to the set X ⇔ x_j = 1);
10 until all rows have been covered;
```

corresponds to data file `data.45` (renamed `S45`), which has been included in the four *Unicost Set Covering* problems, as derived from Steiner's triple systems and accessible on J.E. Beasley's OR-Library site J.E.Beasley [1990]. By setting the values $0, 0.2, \ldots 1$ for $\alpha$ and $1, 2, \ldots 100$ for the seed of the pseudo-random sequence, the results table presented in Table 1.1 has been obtained. This table lists the

| $\alpha \backslash z$ | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | *total* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 0 | 0 | 0 | 0 | 0 | 1 | 9 | 10 | 15 | 17 | 21 | 15 | 88 |
| 0.2 | 0 | 0 | 0 | 1 | 3 | 15 | 34 | 23 | 18 | 5 | 1 | 0 | 100 |
| 0.4 | 0 | 0 | 0 | 5 | 13 | 30 | 35 | 16 | 1 | 0 | 0 | 0 | 100 |
| 0.6 | 0 | 2 | 2 | 45 | 38 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| 0.8 | 0 | 11 | 43 | 46 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| 1.0 | 0 | 55 | 19 | 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |

Table 1.1: Occurrences of solutions by $z$ value for the `S45` instance

number of solutions whose coverage size $z$ lies between 30 and 41. The quality of these solutions is clearly correlated with the value of parameter $\alpha$. For the case $\alpha = 0$ (*random assignment*), it can be observed that the `greedy()` function produces 12 solutions of a size that strictly exceeds 41. No solution with an optimal coverage size of 30 (*known for this instance*) is actually produced.

### 1.4.2 Improvement phase

The improvement algorithm proposed by T. Feo and M.G.C. Resende Feo and Resende [1995] is a simple descent on an elementary neighborhood $\mathcal{N}$. Let $x$ denote the current configuration, then a configuration $x'$ belongs to $N(x)$ if a unique $j$ exists such that $x_j = 1$ and $x'_j = 0$ and moreover that $\forall i \in [1, m] \quad \sum_{j=1}^{n} cover_{ij} \times x'_j \geq 1$. Between two neighboring configurations $x$ and $x'$, a redundant column (from the standpoint of row coverage) was deleted.

**Algorithm 1.4:** `descent(`$x$`)`

**input** : characteristic vector $x$ from the set $X$
**output**: feasible $x$ without any redundant column

```
1 while redundant columns continue to exist do
2     Find redundant j ∈ X such that cost_j is maximized;
3     if j exists then
4         X = X \ {j}
```

Pseudo-code 1.4 describes this descent phase and takes into account the cost of each column, with respect to the column deletion criterion, for subsequent application to the more general Set Covering problem.

Moreover, the same statistical study on the occurrences of the best solutions to the `greedy()` procedure on its own (see Table 1.1) is repeated, this time with addition of the `descent()` procedure, yielding the results provided in Table 1.2. A leftward shift is observed in the occurrences of objective value $z$; such an observation effectively illustrates the benefit of this improvement phase. Before pursuing the various experimental phases, the characteristics of our benchmark will first be presented.

## 1.5 Benchmark

The *benchmark* used for experimentation purposes is composed of fourteen instances made available on J.E. Beasley's `OR-Library` site J.E.Beasley [1990].

The four instances `data.45`, `data.81`, `data.135` and `data.243` (renamed respectively `S45`, `S81`,`S135` and `S243`) make up the test datasets in the reference

| $\alpha \backslash z$ | 30 | 31 | 32 | 33 | 34 | 35 | 36 | 37 | 38 | 39 | 40 | 41 | *total* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0.0 | 0 | 0 | 0 | 0 | 1 | 9 | 10 | 15 | 17 | 21 | 15 | 8 | 96 |
| 0.2 | 0 | 0 | 1 | 3 | 15 | 34 | 23 | 18 | 5 | 1 | 0 | 0 | 100 |
| 0.4 | 0 | 0 | 5 | 13 | 30 | 35 | 16 | 1 | 0 | 0 | 0 | 0 | 100 |
| 0.6 | 2 | 2 | 45 | 38 | 13 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| 0.8 | 11 | 43 | 46 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |
| 1.0 | 55 | 19 | 26 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 100 |

Table 1.2: Improvement in $z$ values for the `S45` instance

| Inst. | $n$ | $m$ | Inst. | $n$ | $m$ | Inst. | $n$ | $m$ |
|---|---|---|---|---|---|---|---|---|
| `G1` | 10000 | 1000 | `H1` | 10000 | 1000 | `S45` | 45 | 330 |
| `G2` | 10000 | 1000 | `H2` | 10000 | 1000 | `S81` | 81 | 1080 |
| `G3` | 10000 | 1000 | `H3` | 10000 | 1000 | `S135` | 135 | 3015 |
| `G4` | 10000 | 1000 | `H4` | 10000 | 1000 | `S243` | 243 | 9801 |
| `G5` | 10000 | 1000 | `H5` | 10000 | 1000 | | | |

Table 1.3: Characteristics of the various instances

article by T. Feo and M.G.C. Resende Feo and Resende [1995]: these are all Unicost Set Covering problems. The ten instances `G1`...`G5` and `H1`...`H5` are considered Set Covering problems. Table 1.3 indicates, for each test dataset, the number $n$ of columns and number $m$ of rows.

The GRASP method was run 100 times for each of the three $\alpha$ coefficient values of $0.1, 0.5$ and $0.9$. Seed $g$ of the `srand(g)` function assumes the values $1, 2, \ldots, 100$. For each method execution, the CPU time is limited to 10 seconds. The computer used for this benchmark is equipped with an $i7$ processor running at 3.4GHz with 8 gigabytes of hard drive memory. The operating system is a Linux, Ubuntu 12.10.

## 1.6 `greedy(`$\alpha$`)+descent` experimentations

Provided below is the pseudo-code of the initial GRASP method version, `GRASP1`, used for experimentation on the fourteen datasets of our *benchmark*.

The `srand()` and `rand()` functions used during the experimental phase are

**Algorithm 1.5:** `GRASP1`

**input** : $\alpha$, random seed seed, time limit.
**output**: $z^{best}$

```
1 srand(seed);
2 z^best ← +∞;
3 repeat
4     x ← greedy(α);
5     x ← descent(x);
6     if z(x) < z^best then
7         z^best ← z(x);
8 until CPU time > time limit;
```

those of the *Numerical Recipes* Press et al. [1992]. Moreover, let's point out that the coding of the $\mathcal{H}$ function is critical: introduction of an incremental computation is essential to obtaining relative short execution times. The values given in Table 1.4 summarize the results output by the `GRASP1` procedure. The primary results

| | | $\alpha = 0.1$ | | | $\alpha = 0.5$ | | | $\alpha = 0.9$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Inst. | $z^*$ | $z$ | # | $\frac{\sum z_g}{100}$ | $z$ | # | $\frac{\sum z_g}{100}$ | z | # | $\frac{\sum z_g}{100}$ |
| G1 | 176 | 240 | 1 | 281.83 | 181 | 1 | 184.16 | 183 | 3 | 185.14 |
| G2 | 154 | 208 | 1 | 235.34 | 162 | 7 | 164.16 | 159 | 1 | 160.64 |
| G3 | 166 | 199 | 1 | 222.59 | 175 | 2 | 176.91 | 176 | 3 | 176.98 |
| G4 | 168 | 215 | 1 | 245.78 | 175 | 1 | 177.90 | 177 | 5 | 178.09 |
| G5 | 168 | 229 | 1 | 249.40 | 175 | 1 | 178.56 | 174 | 6 | 175.73 |
| H1 | 63 | 69 | 1 | 72.30 | 67 | 29 | 67.71 | 67 | 5 | 68.19 |
| H2 | 63 | 69 | 2 | 72.28 | 66 | 1 | 67.71 | 67 | 1 | 68.51 |
| H3 | 59 | 64 | 1 | 68.80 | 62 | 1 | 64.81 | 63 | 34 | 63.66 |
| H4 | 58 | 64 | 1 | 67.12 | 62 | 18 | 62.86 | 63 | 80 | 63.20 |
| H5 | 55 | 61 | 1 | 62.94 | 59 | 2 | 60.51 | 57 | 99 | 57.01 |
| S45 | 30 | 30 | 100 | 30.00 | 30 | 100 | 30.00 | 30 | 100 | 30.00 |
| S81 | 61 | 61 | 100 | 61.00 | 61 | 100 | 61.00 | 61 | 100 | 61.00 |
| S135 | 103 | 104 | 2 | 104.98 | 104 | 4 | 104.96 | 103 | 1 | 104.10 |
| S243 | 198 | 201 | 1 | 203.65 | 203 | 18 | 203.82 | 203 | 6 | 204.31 |

Table 1.4: `greedy(`$\alpha$`)+descent` results

tables provided herein indicate the following:

- the name of the tested instance,
- the best value $z^*$ known for this particular problem, along with,

- for each value of coefficient $\alpha = 0.1, 0.5$ and $0.9$:
  - the best value $z$ found using the GRASP method,
  - the number of times # this value has been reached per 100 runs,
  - the average of the 100 values produced by this algorithm.

For the four instances `S`, the value displayed in column $z^*$ is optimal (Ostrowski et al. [2011]). On the other hand, the optimal value for the other ten instances (`G1`,...,`G5` and `H1`,...,`H5`) remains unknown: the $z^*$ values for these ten instances are the best values published in the literature ( Azimi et al. [2010]; Caprara et al. [1999]; Yagiura et al. [2006]).

With the exception of instance `S243`, the best results are obtained using the values 0.5 and 0.9 of $RCL$ management parameter $\alpha$. For the four instances derived from Steiner's triple problem, the values published by T. Feo and M.G.C. Resende Feo and Resende [1995] are corroborated. However, when compared with the works of Z. Naji-Azimi et al. Azimi et al. [2010], performed in 2010, or even those of A. Caprara et al. Caprara et al. [1998], dating back to 2000, these results prove to be relatively far from the best published values.

## 1.7 Tabu search

This section will focus on adding a Tabu search phase to the GRASP method in order to generate more competitive results with respect to the literature. The algorithm associated with this Tabu search is characterized by:

- an infeasible configuration space $\mathcal{S}$, such that $z(x) < z^{min}$,
- a simple move (of the *1-change*) type,
- a strict Tabu list.

### 1.7.1 The search space

In relying on the configuration $x^0$ output by the descent phase (corresponding to a set $X$ of columns guaranteeing row coverage), the Tabu search will explore the space of configurations $x$ with objective value $z(x)$ less than $z^{min} = z(x^{min})$, where $x^{min}$ is the best feasible solution found by the algorithm. The search space $\mathcal{S}$ is thus formally defined as follows:

$$\mathcal{S} = x \in \{0,1\}^n \,/\, z(x) < z(x^{min})$$

### 1.7.2 Evaluation of a configuration

It is obvious that the row coverage constraints have been relaxed. The $\mathcal{H}$ evaluation function of a column $j$ now contains two components:

$$\mathcal{H}_1(j) = \begin{cases} \mathcal{C}(X \cup \{j\}) - \mathcal{C}(X) & \text{if } x_j = 0 \\ \mathcal{C}(X \setminus \{j\}) - \mathcal{C}(X) & \text{if } x_j = 1 \end{cases}$$

and

$$\mathcal{H}_2(j) = \begin{cases} cost_j & \text{if } x_j = 0 \\ -cost_j & \text{if } x_j = 1 \end{cases}$$

This step consists of repairing the coverage constraints (i.e. maximizing $\mathcal{H}_1$) at the lowest cost (minimizing $\mathcal{H}_2$).

### 1.7.3 Managing the Tabu list

This task involves use of the *Reverse Elimination Method* proposed by F. Glover and M. Laguna (Glover and Laguna [1997]), which has been implemented in order to exactly manage the Tabu status of potential moves: a move is forbidden if and only if it leads to a previously encountered configuration. This Tabu list is referred to as a strict list.

**Algorithm 1.6:** `updateTabu(`$j$`)`

```
   input : j ∈ [0, n − 1]
1  running list[iter] = j;
2  i ← iter;
3  iter ← iter + 1;
4  repeat
5  |  j ← running list[i];
6  |  if j ∈ RCS then
7  |  |  RCS ← RCS/{j};
8  |  else
9  |  |  RCS ← RCS ∪ {j};
10 |  if |RCS| = 1 then
11 |  |  j = RCS[0] is tabu;
12 |  i ← i − 1
13 until i < 0;
```

The algorithm we describe herein is identical to that successfully run on another combinatorial problem with binary variables Nebel [2001]. The *running list* is actually a table in which a recording is made, upon each iteration, of the column $j$ targeted by the most recent move: $x_j = 0$ or $x_j = 1$. This column is considered the move attribute. The $RCS$ (for *Residual Cancellation Sequence*) is another table in which attributes will be either added or deleted. The underlying principle consists of reading one by one, from the end of the *running list*, past move attributes, in adding $RCS$ should they be absent and removing $RCS$ if already present. The following equivalence is thus derived: $|RCL| = 1 \Leftrightarrow RCL[0]$ prohibited. The interested reader is referred to the academic article by F. Dammeyer and S. Voss Dammeyer and Voß [1993] for further details on this specific method.

### 1.7.4 Neighborhood

We have made use of an elementary *1-change* move: $x' \in \mathcal{N}(x)$ if $\exists! j / x'_j \neq x_j$. The neighbor $x'$ of configuration x only differs by one component yet still satisfies the condition $z(x') < z^{min}$, where $z^{min}$ is the value of the best feasible configuration identified. Moreover, the chosen non-Tabu column $j$ minimizes the hierarchical criterion $((\mathcal{H}_1(j), \mathcal{H}_2(j)))$. Pseudo-code 1.7 describes the evaluation function for this neighborhood.

**Algorithm 1.7:** `eval`$\mathcal{H}(j1, j2)$

**input** : column interval $[j1, j2]$
**output**: *best* column identified $j^*$

```
1  j* ← −1;
2  H1* ← −∞;
3  H2* ← +∞;
4  for j1 ≤ j ≤ j2 do
5      if j non tabu then
6          if (x_j = 1) ∨ (z + cost_j < z^min) then
7              if (H1(j) > H1*) ∨ (H1(j) = H1* ∧ H2(j) < H2*) then
8                  j* ← j;
9                  H1* ← H1(j);
10                 H2* ← H2(j);
```

### 1.7.5 The Tabu algorithm

The general `Tabu()` procedure uses as an argument the solution $x$ produced by the `descent()` procedure, along with a maximum number of iterations $N$. Rows 6 through 20 of Algorithm 1.8 correspond to a search diversification mechanism. Each time a feasible configuration is produced (i.e. $|X| = m$), the value $z^{min}$ is updated and the Tabu list is reset to zero.

The references to rows 2 and 20 will be helpful in explaining the algorithm in Section 1.9.

**Algorithm 1.8:** `tabu(`$x$`,`$N$`)`

**input** : feasible solution $x$, number of iterations $N$
**output**: $z^{min}$, $x^{min}$

```
2  z^min ← z(x) ;
3  iter ← 0 ;
4  repeat
6      r ← rand() modulo n;
8      j* ← evalH(r, n − 1);
10     if j* < 0 then
12         j* ← evalH(0, r − 1);
13     if x_{j*} = 0 then
14         add column j*;
15     else
16         remove column j*;
17     if |X| = m then
18         z^min ← z(x) ;
20         x^min ← x ;
21         iter ← 0;
22         delete the Tabu status ;
23     updateTabu(j*);
24 until iter ≥ N or j* < 0;
```

## 1.8 `greedy(`$\alpha$`)+descent+Tabu` experimentations

For this second experimental phase, the benchmark is similar to that discussed in Section 1.6. Total CPU time remains limited to 10 seconds, while the maximum number of iterations without improvement for the `Tabu()` procedure equals half the number of columns for the treated instance (i.e.$n/2$). The pseudo-code of the `GRASP2` procedure is specified by Algorithm 1.9.

Table 1.5 effectively illustrates the significant contribution of the Tabu search to the GRASP method. All $z^*$ column values are found using this version of the GRASP method. In comparison with Table 1.4, parameter $\alpha$ is no longer seen to exert any influence on results. It would seem that the *multi-start* function of the GRASP method is more critical to the Tabu phase than control over the $RCL$ candidate list. However, as will be demonstrated in the following experimental phase, it still appears that rerunning the method, under parameter $\alpha$ control, does play a determinant role in obtaining the best results.

**Algorithm 1.9:** `GRASP2`

**input** : $\alpha$, random seed *seed*, time limit.
**output**: $z^{best}$

```
1 z^best ← +∞;
2 srand(seed);
3 repeat
4 |   x ← greedy(α);
5 |   x ← descent(x);
6 |   z ← Tabu(x, n/2);
7 |   if z < z^best then
8 |   |   z^best ← z;
9 until CPU time > time limit;
```

| | | $\alpha = 0.1$ | | | $\alpha = 0.5$ | | | $\alpha = 0.9$ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Inst. | $z^*$ | $z$ | # | $\frac{\sum z_g}{100}$ | $z$ | # | $\frac{\sum z_g}{100}$ | z | # | $\frac{\sum z_g}{100}$ |
| G1 | 176 | 176 | 100 | 176.00 | 176 | 96 | 176.04 | 176 | 96 | 176.04 |
| G2 | 154 | 154 | 24 | 154.91 | 154 | 32 | 155.02 | 154 | 57 | 154.63 |
| G3 | 166 | 167 | 4 | 168.46 | 167 | 10 | 168.48 | 166 | 1 | 168.59 |
| G4 | 168 | 168 | 1 | 170.34 | 170 | 35 | 170.77 | 170 | 29 | 170.96 |
| G5 | 168 | 168 | 10 | 169.59 | 168 | 7 | 169.66 | 168 | 10 | 169.34 |
| H1 | 63 | 63 | 11 | 63.89 | 63 | 2 | 63.98 | 63 | 5 | 63.95 |
| H2 | 63 | 63 | 21 | 63.79 | 63 | 13 | 63.87 | 63 | 5 | 63.95 |
| H3 | 59 | 59 | 76 | 59.24 | 59 | 82 | 59.18 | 59 | 29 | 59.73 |
| H4 | 58 | 58 | 99 | 58.01 | 58 | 98 | 58.02 | 58 | 100 | 58.00 |
| H5 | 55 | 55 | 100 | 55.00 | 55 | 100 | 55.00 | 55 | 100 | 55.00 |
| S45 | 30 | 30 | 100 | 30.00 | 30 | 100 | 30.00 | 30 | 100 | 30.00 |
| S81 | 61 | 61 | 100 | 61.00 | 61 | 100 | 61.00 | 61 | 100 | 61.00 |
| S135 | 103 | 103 | 49 | 103.51 | 103 | 61 | 103.39 | 103 | 52 | 103.48 |
| S243 | 198 | 198 | 100 | 198.00 | 198 | 100 | 198.00 | 198 | 100 | 198.00 |

Table 1.5: `greedy(`$\alpha$`)+descent+Tabu` results

## 1.9 `greedy(1)+Tabu` experimentations

To confirm the benefit of this GRASP method, let's now observe the behavior of Algorithm 1.10: `TABU`. For each value of the pseudo-random function `rand()` seed ($1 \leq g \leq 100$ for the call-up of `srand()`), a solution is built using the `greedy(1)` procedure, whereby redundant $x$ columns are deleted in allowing for completion of the `Tabu(`$x$`,`$n$`)` procedure, provided CPU time remains less than 10 seconds.

| Inst. | $z$ | # | $\frac{\sum z_g}{100}$ | Inst. | $z$ | # | $\frac{\sum z_g}{100}$ | Inst. | $z$ | # | $\frac{\sum z_g}{100}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| G1 | 176 | 95 | 176.08 | H1 | 63 | 2 | 63.98 | S45 | 30 | 100 | 30.00 |
| G2 | 154 | 24 | 155.22 | H2 | 63 | 4 | 63.96 | S81 | 61 | 100 | 61.00 |
| G3 | 167 | 19 | 168.48 | H3 | 59 | 36 | 59.74 | S135 | 103 | 28 | 103.74 |
| G4 | 170 | 3 | 171.90 | H4 | 58 | 91 | 58.09 | S243 | 198 | 98 | 198.10 |
| G5 | 168 | 20 | 169.39 | H5 | 55 | 97 | 55.03 | | | | |

Table 1.6: `greedy(1)+descent+Tabu` results

For this final experimental phase, row 2 has been replaced in pseudo-code 1.8 by $z^{min} \leftarrow +\infty$. Provided the CPU time allocation has not been depleted, the `Tabu()` procedure is reinitiated starting with the best solution it was able to produce during the previous iteration. This configuration is saved in row 20. Moreover, the size of the *running list* is twice as long.

**Algorithm 1.10:** `TABU`

**input** : random seed, time limit.
**output**: $z^{best}$

```
1  z^best ← +∞;
2  srand(seed);
3  x ← greedy(1);
4  x^min ← descent(x);
5  repeat
6  |   x ← x^min;
7  |   z, x^min ← Tabu(x, n);
8  |   if z < z^best then
9  |   |   z^best ← z;
10 until CPU time > time limit;
```

In absolute value terms, these results fall short of those output by Algorithm 1.9: `GRASP2`. This `TABU` version has produced values of 167 and 170 for instances `G3` and `G4` vs. 166 and 168 respectively for the `GRASP2` version. Moreover, the majority of average values are of poorer quality than those listed in Table 1.5.

## 1.10 Conclusion

This chapter has presented the principles behind the GRASP method and has detailed their implementation in the aim of resolving large-sized instances associated with a hard combinatorial problem. Section 1.4.1 exposed the simplicity involved in modifying the greedy heuristic proposed by T.A. Feo and M.G.C. Resende, namely:

$$\mathcal{H}(j) = \begin{cases} \mathcal{C}(X \cup \{j\}) - \mathcal{C}(X) & \text{if } x_j = 0 \\ \mathcal{C}(X \setminus \{j\}) - \mathcal{C}(X) & \text{if } x_j = 1 \end{cases}$$

in order to take into account the column cost and apply the construction phase, not only to the minimum coverage problem, but to the Set Covering Problem as well.

The advantage of enhancing the improvement phase has also been demonstrated by adding, to the general GRASP method loop, a Tabu search on an elementary neighborhood.

# Chapter 2

# One-dimensional Bin Packing

In this chapter we study the One-dimensional Bin Packing problem (BPP) and present efficient and effective local search algorithm for solving it.

## 2.1 Introduction

Given a set $I = \{1, 2, \ldots, n\}$ of items with associated weights $w_i$ $(i = 1, \ldots, n)$, the bin packing problem (BPP) consists of finding the minimum number of bins, of capacity $C$, necessary to pack all the items without violating any of the capacity constraints. In other words, one has to find a partition of items $\{I_1, I_2, \ldots, I_m\}$ such that

$$\sum_{i \in I_j} w_i \leq C, \quad j = 1, \ldots, m$$

and $m$ is minimum. The bin packing problem is known to be NP-hard (Garey and Johnson [1979]). One of the most extensively studied combinatorial problems, BPP has a wide range of practical applications such as in storage allocation, cutting stock, multiprocessor scheduling, loading in flexible manufacturing systems and many more. The Vector Packing problem (VPP) is a generalization of BPP with multiple resources. Item weights $w_i^r$ and bin capacity $C_r$ are given for each resource $r \in \{1, \ldots, R\}$ and the following constraint has to be respected:

$$\sum_{i \in I_j} w_i^r \leq C_r, \quad r = 1, \ldots, R, j = 1, \ldots, m$$

Bin packing problem with cardinality constraints (BPPC) is a bin packing problem where, in addition to capacity constraints, an upper bound $k \geq 2$ on the number of items packed into each bin is given. This constraint can be expressed as:

$$\sum_{i \in I_j} 1 \leq k, \quad j = 1, \ldots, m$$

Obviously, bin packing with cardinality constraints can be seen as a two dimensional vector packing problem where $C_2 = k$ and $w_i^1 = w_i^2 = 1$ for each item $i \in 1, \ldots, n$.
Without loss of generality we can assume that capacities and weights are integer in each of the defined problems.

We will present a new improvement heuristic based on a local search for solving BPP, VPP with two resources (2-DVPP) and BPPC. The method will first be described in detail for a BPP problem, followed by the set of underlying adaptations introduced to solve the 2-DVPP and BPPC.

A possible mathematical formulation of BPP is

$$\text{minimize } z = \sum_{i=1}^{n} y_i \tag{2.1}$$

$$\text{subject to } z = \sum_{j=1}^{n} w_j x_{ij} \leq C y_i, i \in N = \{1, 2, \ldots, n\} \tag{2.2}$$

$$\sum_{i=1}^{n} x_{ij} = 1, j \in N \tag{2.3}$$

$$y_i \in \{0, 1\}, i \in N \tag{2.4}$$

$$x_{ij} \in \{0, 1\}, i, j \in N, \tag{2.5}$$

where
$y_i = 1$ if bin $i$ is used, otherwise $y_i = 0$;
$x_{ij} = 1$ if item $j$ is assigned to bin $i$, otherwise $x_{ij} = 0$.

**Definition 1** *For a subset $S \subseteq I$, we let* $\mathrm{w}(S) = \sum_{i \in S} w_i$.

Some of the most simple algorithms for solving BPP are given in the following discussion.

*Next Fit Heuristic (NF)* works as follows: Place the items in the order in which they arrive. Place the next item into the current bin if it fits. Otherwise, create a new bin. Pseudo code is given in Algorithm 2.1.

This algorithm can waste a lot of bin space, since the bins we close may not be very full. However, it does not require memory of any bin except the current one. One should be able to improve the performance of the algorithm by considering previous bins that might not be full. Similar to NF are *First Fit Heuristic (FF), Best Fit Heuristic (BF), and Worst Fit Heuristic (BF). First Fit* works as follows: Place the items in the order in which they arrive. Place the next item into the lowest numbered bin in which it fits. If it does not fit into any open bin, create a new bin. *Best Fit and Worst Fit heuristics* are similar to FF, but instead of placing an item into the first available bin, the item is placed into the bin with smallest (greatest for WF) remaining capacity it fits to.

If it is permissible to preprocess the list of items, significant improvements are

**Algorithm 2.1:** Next Fit

```
1  Input: A set of all items I, w_i ≤ C, ∀i ∈ I;
2  Output: A partition {B_i} of I where w(B_i) ≤ C for each i;
3  b ← 0;
4  for each i ∈ I do
5      if w_i + w(B_b) ≤ C then
6          B_b ← B_b ∪ {i}
7      else
8          b ← b + 1;
9          B_b ← {i};
10 return B_1, ..., B_b;
```

possible for some of the heuristic algorithms. For example, if the items are sorted before they are packed, a decreasing sort improves the performance of both the First Fit and Best Fit algorithms. This two algorithms are refered to as *First Fit Decreasing (FFD)* and *Best Fit Decreasing*. Johnson [1973] showed that $FFD$

heuristic uses at most $11/9 \times OPT + 4$ bins, where $OPT$ is the the optimal solution (smallest number of bins) to the problem.

In our proposed heuristic, the solution is iteratively improved by decreasing the number of bins being utilized. The procedure works as follows. First, the upper bound on the solution value, $UB$, is obtained by a variation of First Fit (FF) heuristic. Next, an attempt is made to find a feasible solution with $UB - 1$ bins, and this process continues until reaching lower bound, the time limit or maximum number of search iterations. Aside from the simple lower bound, $\left\lceil \frac{\sum_{i=1}^{n} w_i}{C} \right\rceil$, lower bounds developed by Fekete and Schepers [2001], Martello and Toth [1990] (bound $L_3$) and Alvim et al. [2004] have also been used.

In order to find a feasible solution with a given number of bins, $m < UB$, a local search is employed. As opposed to the majority of work published on BPP, a local search explores partial solutions that consist of a set of assigned items without any capacity violation and a set of non-assigned items. Moves consist of rearranging both the items assigned to a single bin and non-assigned items, i.e. adding and dropping items to and from the bin. The objective here is to minimize the total weight of non-assigned items. This local search on a partial configuration is called the Consistent Neighborhood Search and has been proven efficient in several combinatorial optimization problems (Habet and Vasquez [2004]; Vasquez et al. [2003]). Therefore, our approach will be refered as CNS_BP in the reminder of the chapter.

An exploration of this search space of partial solutions comprises two parts, which will be run in succession: 1) a tabu search with limited add/drop moves and 2) a descent with a general add/drop move. This sequence terminates when a complete solution has been found or the running time limit (or maximum number of iterations) has been exceeded.
Additionally, the algorithm makes use of a simple reduction procedure that consists of fixing the assignments of all pairs of items that fill an entire bin. More precisely, s set of item pairs $(i, j)$ such that $w_i + w_j = C$ has been identified, and the problem is now reduced by deleting these items (or setting their assignments).

This same reduction has been used in the majority of papers on BPP. It is important to mention that not using reduction procedure will not have a significant influence on the final results (but can speed up the search) and, moreover, no reduction is possible for a big percentage of the instances considered.

This chapter will be organized as follows. Section 2.2 will address relevant work, and our approach will be described in Section 2.3. The general framework will be presented first, followed by a description of all algorithmic components. A number of critical remarks and parameter choices will be discussed in Section 2.4. Section 2.5 presents a summary of methodological adaptations to 2-DVPP and BPPC. The results of extensive computational experiments performed on the available set of instances, for BPP, 2-DVPP and BPPC will be provided in Section 2.6, followed by conclusions drawn in the final section.

## 2.2 Relevant work

### 2.2.1 BPP

A large body of literature relative to one-dimensional bin packing problems is available. Both exact and heuristic methods have been applied to solving the problem. Martello and Toth [1990] proposed a branch-and-bound procedure (MTP) for solving the BPP. Scholl et al. [1997] developed a hybrid method (BISON) that combines a tabu search with a branch-and-bound procedure based on several bounds and a new branching scheme. Schwerin and Wäscher [1999] offered a new lower bound for the BPP based on the cutting stock problem, then integrated this new bound into MTP and achieved high-quality results. de Carvalho [1999] proposed an exact algorithm using column generation and branch-and-bound.

Gupta and Ho [1999] presented a minimum bin slack (MBS) constructive heuristic. At each step, a set of items that fits the bin capacity as much as possible is identified and packed into the new bin. Fleszar and Hindi [2002] developed a hybrid algorithm that combines a modified version of the MBS and the Variable Neighborhood Search. Their hybrid algorithm performed very well in computational experiments, having obtained the optimal solution for 1329 out of the 1370

instances considered.

Alvim et al. [2004] presented a hybrid improvement heuristic (HI_BP) that uses tabu search to move the items between bins. In their algorithm, a complete yet infeasible configuration is to be repaired through a tabu search procedure. Simple "shift and swap" neighborhoods are explored, in addition to balancing/unbalancing the use of bin pairs by means of solving a Maximum Subset Sum problem. HI_BP performed very well, as evidenced by finding optimal solutions for all 1370 benchmark instances considered by Fleszar and Hindi [2002] and a total of 1582 out of the 1587 optimal solutions on an extensively studied set of benchmark instances.

In recent years, several competitive heuristics have been presented with results similar to those obtained by HI_BP. Singh and Gupta [2007] proposed a compound heuristic (C_BP), in combining a hybrid steady-state grouping genetic algorithm with an improved version of Fleszar and Hindi's Perturbation MBS. Loh et al. [2008] developed a weight annealing (WA) procedure, by relying on the concept of weight annealing to expand and accelerate the search by creating distortions in various parts of the search space. The proposed algorithm is simple and easy to implement; moreover, these authors reported a high quality performance, exceeding that of the solutions obtained by HI_BP.
Fleszar and Charalambous [2011] offered a modification to the Perturbation-MBS method (Fleszar and Hindi [2002]) that introduces a new sufficient average weight (SAW) principle to control the average weight of items packed in each bin (referred to as Perturbation-SAWMBS). This heuristic has outperformed the best state-of-the-art HI_BP, C_BP and WA algorithms. Authors also presented corrections to the results that were reported for the WA heuristic, obtaining significantly lower quality results comparing to those reported in Loh et al. [2008].

To the best of our knowledge, the most recent work in this area, presented by (Quiroz-Castellanos et al. [2015]), entails a grouping genetic algorithm (GGA-CGT) that outperforms all previous algorithms in terms of number of optimal solutions found, particularly with a set of most difficult instances `hard28`.
Brandão and Pedroso [2013] devised an exact approach for solving bin packing and cutting stock problems based on an Arc-Flow Formulation of the problem;

these authors made use of a commercial Gurobi solver to process their model. They were able to optimally solve all standard bin packing instances within a reasonable computation time, including those instances that have not been solved by any heuristic method.

### 2.2.2 VPP

As regards the two-dimensional VPP, Spieksma [1994] proposed a branch-and-bound algorithm, while Caprara and Toth [2001] forwarded exact and heuristic approaches as well as a worst-case performance analysis. A heuristic approach using set-covering formulation was presented by Monaci and Toth [2006]. Masson et al. [2013] proposed an iterative local search (ILS) algorithm for solving the Machine Reassignment Problem and VPP with two resources; they reported the best results on the classical VPP benchmark instances of Spieksma (1994) and Caprara and Toth (2001).

## 2.3 A proposed heuristic

This section will describe our improvement heuristic. General improvement process is given in Algorithm 2.2. Algorithm starts with applying a simple reduction procedure and constructing initial (feasible and complete) solution by applying FF heuristic on a randomly sorted set of items. This initial solution, containing $UB$ bins, is then to be improved by a local search based procedure, which represents the core element of our proposal. More precisely, an attempt is made to find a complete solution with $m = UB - 1$ bins by applying local search on a partial solution, and this process continues until reaching lower bound, the time limit or maximum number of iterations (precisely defined later).

The remainder of a section will describe a procedure aimed at finding a feasible solution with a given number of bins, $m$. The inherent idea here is to build a partial solution with $m - 2$ bins and then transform it into a complete feasible solution with $m$ bins through applying a local search. The partial solution is one that contains a set of items assigned to $m-2$ bins, without any capacity violation, and

**Algorithm 2.2:** $CNS\_BP$

```
remove item pairs (i, j) such that w_i + w_j = C;
compute lower bound LB;
random shuffle the set of items;
m ← upper bound by First Fit;
while m > LB ∧ time limit not exceeded do
    m ← m − 1;
    create partial solution S with m − 2 bins;
    CNS(S);
return the last complete solution;
```

a set of non-assigned items. The goal of the local search is, by rearranging the items, to obtain a configuration such that non-assigned items can be packed into two bins, thus producing a feasible solution with $m$ bins.

One can notice that termination of the search by finding a complete solution, i.e. packing non-assigned items into two bins is not possible if more than two "big" items (with weight greater than or equal to half of the bin capacity) are non-assigned. Therefore, maximum number of non-assigned big items is limited to two during the whole procedure. When packing non-assigned items into two bins is possible, complete solution is obtained by simply adding the two new bins to the current set of bins.

Partial solution with $m - 2$ bins is built by deleting three bins from the last complete solution i.e. by removing all the items from these three bins and adding them to the set of non-assigned items (note that last feasible solution contains $m + 1$ bins). Bins to be deleted are selected in the following way:

- select the last two bins from a complete solution,
- select the last bin (excluding the last two) such that total number of non-assigned "big" items does not exceed two.

The capacity of all bins is never violated at any time during the procedure. Items have been randomly sorted before applying FF in order to avoid solutions with many small or many big non-assigned items, which could make the search more difficult or slower (this is the case, for example, if items are sorted in decreasing order). This very same procedure is then used for all types of instances,

e.g. the same initial solution, same parameters, same order of neighborhood exploration.

For the sake of simplicity, let's assume that the non-assigned items are packed into the special bin with unlimited capacity, called *trash can* and denoted by $TC$. Let $B = \{b_1, b_2, \ldots, b_{m-2}\}$ be the set of currently utilized bins, $I_B \subseteq I$ the set of items assigned to the bins in $B$ and $I_b$ the set of items currently packed into bin $b \in B$. Analogously, let $I_{TC}$ denote the set of currently non-assigned items. Total weight and cardinality of a set of items $S$ will be denoted by $w(S)$ and $|S|$ respectively. For the sake of simplicity, total weight and current number of items currently assigned to bin $b \in B \cup TC$ will be denoted by $w(b) = w(I_b)$ and $|b| = |I_b|$.

### 2.3.1 Local Search

The Local Search procedure is applied to reach a complete solution with $m$ bins, in starting from a partial one with $m-2$ bins constructed as described before. Several neighborhoods are explored during the search, which consists of two procedures executed in succession until a stopping criterion is met. These two procedures are: a) tabu search procedure and b) hill climbing/descent procedure. All moves consist of swapping the items between a bin in $B$ and trash can $TC$.
Formally speaking, the local search moves include:

1. $Swap(p, q)$ - consists of swapping $p$ items from a bin $b \in B$ with $q$ items from $TC$,

2. $Pack(b)$ - consists of optimally rearranging the items between bin $b \in B$ and trash can $TC$, such that the remaining capacity in $b$ is minimized, whereby a set of items (packing) $P \subseteq I_b \cup I_{TC}$ that fits the bin capacity as optimally as possible is determined. $Pack$ is a generalization of a $Swap$ move with $p$ and $q$ both being unlimited.

Only the moves not resulting in any capacity violation are considered during this search. Swap move is used only in the tabu search procedure, while descent procedure makes use of a Pack move exclusively. Pseudo code of the procedure is given in Algorithm 2.3 and two main parts, TabuSearch() and Descent() procedures, will

be explained below in the corresponding sections.

Before explaining each of the two main parts of the local search, we will discuss

**Algorithm 2.3:** $CNS()$

```
1 input: partial solution currSol;
2 while time or iterations limit not exceeded and complete solution not found do
3 |   currSol ← TabuSearch(currSol);
4 |   currSol ← Descent(currSol);
5 return currSol;
```

the search neighborhoods, objective function and search termination conditions. The goal of the local search procedure is to optimize the following lexicographic fitness function:

1. minimize the total weight on non-assigned items (minimize use of the trash can) : min $w(TC)$;

2. maximize the number of items in the trash can: max $|TC|$.

The first objective is quite natural, while the second one is introduced in order to yield items with lower weights in the trash can, as this could: 1) increase the chance of terminating the search; and/or 2) enable a wider exploration of the search space. Formally, the following fitness function is to be minimized

$$obj(TC) = n \times w(TC) - |TC| \tag{2.6}$$

The maximum number of items from the same bin that can be rearranged in a single $Swap$ move is limited to three. More precisely, $Swap(p, q)$ moves with

$$(p, q) \in \{(0, 1), (1, 1), (2, 1), (1, 2), (2, 2), (2, 3), (3, 2)\}$$

have been considered. $Swap(0, 1)$ corresponds to $shift$ move, which consists of shifting (or adding) the item from the trash can to bin $b \in B$. All other possible moves such as $Swap(1, 0)$, $Swap(0, 2)$ and $Swap(3, 1)$ have been omitted since they increase the complexity of the neighborhood evaluation without improving the final results. The results obtained by not allowing more than two items from

a single bin to be swapped ($(p, q) \in \{(0, 1), (1, 1), (2, 1), (1, 2), (2, 2)\}$) will also be reported. Note that the higher complexity of these $Swap(p, q)$ moves, with respect to the classical shift and swap moves used in the literature, is compensated by the fact that no moves between pairs of bins in $B$ are performed.

Generating optimal packing for a set of items is a common procedure introduced in several papers (Gupta and Ho [1999], Fleszar and Hindi [2002], Fleszar and Charalambous [2011]) and originally proposed in Gupta and Ho [1999]. The *Pack* move is the same as the "load unbalancing" used in Alvim et al. [2004]. A Packing problem is equivalent to the Maximum Subset Sum (MSS) problem and can be solved exactly, for instance by either dynamic programming or enumeration. Let's note that the packing procedure is only being used for a small subset of items, i.e. the set of items belonging to a single bin $b \in B$ or trash can $TC$. Let $pack\_set(S)$ denote the solution to the MSS problem, which is a feasible subset $P \subseteq S$ (whose sum of weights does not exceed $C$) with maximum total weight. The enumeration procedure has been used herein to solve the packing problem and pseudo code is given in Algorithm 2.4. Clearly, the complexity of the enumeration procedure is $\mathcal{O}(2^l)$, where $l$ is a number of considered items. The structure of the available instances makes this approach reasonable, though a simple dynamic programming procedure of complexity $\mathcal{O}(l \times C)$, can, if necessary, also be used.

As mentioned before, no more than two "big" items (with weights greater than or equal to $C/2$) are allowed to be assigned to the trash can during the entire solving procedure. This is easily achieved by forbidding all Swap and Pack moves that result in having three or more big items in the trash can, but is omitted in the presented algorithms (pseudo-codes) for the simplicity reasons.

During the search, each time the total weight in the trash can is less than or equal to $2C$, an attempt is undertaken to pack all items from the trash can into the two bins. This step involved simply uses the same *Pack* procedure, with quite

**Algorithm 2.4:** *pack_set*()

```
1 input: set of items S = {i_1, ..., i_k}, current packing P (initialized to empty set);
2 output: best packing P* (initialized to empty set);
3 if S ≠ ∅ then
4     if w(P) + w_{i_1} ≤ C then
5         pack_set(S \ {i_1}, P ∪ {i_1});
6     pack_set(S \ {i_1}, P);
7 else
8     if (w(P) > w(P*)) ∨ (w(P) = w(P*) ∧ |P| < |P*|) then
9         P* ← P
```

obviously packing into two bins being possible (feasible) if and only if

$$w(pack\_set(I_{TC})) \geq w(TC) - C. \tag{2.7}$$

If packing into two bins is indeed possible, then the procedure terminates. Aside from the lower bound and time limit termination criteria, the procedure terminates when total number of solutions with $w(TC) \leq 2C$ obtained during the search exceeds a given number. Terminating the search after failing to pack non-assigned items into two bins too many times seems to be reasonable and this limit is set to 100000 for all considered instances. On the other hand, further exploration of the search space does not look promising if solution with $w(TC) \leq 2C$ cannot be obtained in a reasonable time. Therefore, we also decide to terminate the search if no solution with $w(TC) \leq 2C$ has been found during the first ten algorithm loops (Tabu + Descent).

#### 2.3.1.1 Tabu search

The main component of the improvement procedure is a tabu search that includes *Swap* moves between trash can and bins in $B$. In each iteration of the search, all swap moves between trash can and each bin have been evaluated and the best non-tabu move relative to the defined objective ( minimizing trash can use) is performed. Should two or more moves with the same objective exist, then a random choice is made. Note that the best move is carried out even if it does not improve

the solution with respect to the objective function. Each time the total weight of items in the trash can remains less than or equal to $2C$, an attempt is made to terminate the search by packing non-assigned items into two bins.
This process repeats until no feasible and non-tabu move exists or until the time limit $timeLimitTabu$ has been exceeded or the maximum number of moves without improvement ($maxNmbIters$) has been reached.
Running time of the tabu search has been limited by $timeLimitTabu$; all results reported here were obtained with a one-second limit. The maximum number of iterations without improvement in a tabu search has been set to $|B| \times |I|$.

Whenever an item with weight $w$ is placed into bin $b$ via a swap move, all swap moves that include an item from $b$ with weight $w$ become tabu for a specific number of iterations. Only removing the items from the bin and placing them into the trash can may then be considered tabu, implying that moving an item from the trash can to a bin is never tabu. The number of iterations for which moving the item of weight $w$ from bin $b$ to the trash can is tabu depends on the frequency of assigning an item of the same weight $w$ to the same bin, i.e. on the number of swap moves performed that place the item of weight $w$ into $b$, $freq(b, w)$. More precisely, the given move is tabu for $freq(b, w)/2$ iterations.

Given that the objective of a local search, as defined above, is lexicographic, minimizing the total weight of non-assigned items has a higher priority than maximizing the number of non-assigned items. Nevertheless, this objective can lead the search to the configurations with quite good first objective, but very low number of non-assigned items, which might make the termination and exploration of the search more difficult. We have therefore decided to rely on two different variants of the tabu search procedure, namely:

- tabu search consisting of all defined $Swap(p, q)$ moves,
- tabu search consisting of subset of moves that do not decrease the second objective i.e. with $p \geq q$ ($\Rightarrow (p, q) \in \{(1, 1), (2, 1), (2, 2), (3, 2)\}$)

Two variants differ only in the set of allowed swap moves (line 7 in Algorithm 2.5) and are applied one after another, meaning that the whole tabu search procedure

consists of sequentially calling these two variants. Introducing the second variant of the tabu search has significantly improved the results obtained on `hard28` dataset (around 5 new optimal solutions in average).
Tabu search procedure returns either the best found solution in case initial solution is improved or the last obtained solution (see the last 5 lines in Algorithm 2.5).

**Algorithm 2.5:** *TabuSearch*()

```
1  input: current solution initTabuSol;
2  maxIters ← |B| × |I|;
3  iter ← 0;
4  bestObj ← current objective;
5  bestSol ← current solution;
6  while iter < maxIters ∧ time limit timeLimitTabu not exceeded do
7      evaluate all Swap(p, q) moves between each bin b and TC ;
8      perform a non-tabu swap move that minimizes obj(TC) (choose the random one in
       case more equal moves exist);
9      if current objective < bestObj then
10         bestSol ← current solution;
11         bestObj ← current objective;
12         iter ← 0;
13         reset tabu;
14     if w(TC) ≤ 2C ∧ w(pack_set(I_TC)) ≥ w(TC) − C then
15         TERMINATE;
16     update tabu list;
17     iter ← iter + 1;
18 if bestSol ≠ initTabuSol then
19     return bestSol;
20 else
21     return current solution;
```

#### 2.3.1.2 Descent procedure

The second part of the search procedure involves exploring the search space by applying *Pack* move, which is exclusively reserved as part of the descent (hill climbing) procedure due to the greater complexity associated with this move. Like in the tabu procedure, this move is performed only between trash can and bin in $B$. The $Pack(b)$ move is executed for each bin $b \in B$ until no improvement in the objective can be achieved. Formally, $Pack(b)$ consists of assigning a set of items $pack\_set(I_b \cup I_{TC})$ to bin $b$ and set $(I_B \cup I_{TC}) \setminus pack\_set(I_b \cup I_{TC})$ to trash

can. The order of bins in $B$ while exploring the neighborhood is random. It is clear that the defined objective cannot increase during the procedure due to the nature of the pack move. As in the tabu search procedure, a feasible packing of non-assigned items into two bins is attempted after each $Pack$ move which results in $w(TC) \leq 2C$. The running time of the descent procedure is significantly less than that of the tabu search, which is understandable when taking into account the fact that all moves performed are improvements and moreover that the move complexity is not much greater.

Complete $Pack(b)$ move is performed only when total number of considered items ($|I_b \cup I_{TC}|$) is not greater than 20. Nevertheless, a limited $Pack(b)$ move considering a random subset of items containing no more than ten items from $b$ and no more than ten items from $TC$ is performed. Thus, complexity of the $Pack$ move never exceeds $2^{20}$. Descent procedure is listed in Algorithm 2.6.

**Algorithm 2.6:** *Descent*()

```
1 repeat
2     randomly sort the set of bins B;
3     for each b ∈ B do
4         perform Pack move between b and TC: Pack(b);
5         if objective improved ∧ w(TC) ≤ 2C ∧ w(pack_set(I_TC)) ≥ w(TC) − C then
6             TERMINATE;
7 until no objective improvement made;
```

## 2.4 Discussion and parameters

Unlike many published algorithms, only one procedure has been implemented to build initial solution. We decided to use FF heuristic because of its simplicity, while randomly sorting the set of items before FF is used in order to avoid search configurations with many items of similar sizes (too many small or too many big items for example). The trash can could contain too many small items if, for example, First Fit Decrease heuristic is used. Furthermore, solutions obtained by using only First Fit heuristic on a given set of items (without random shuffling) depend on the order of items given in the benchmark data files; this can be a

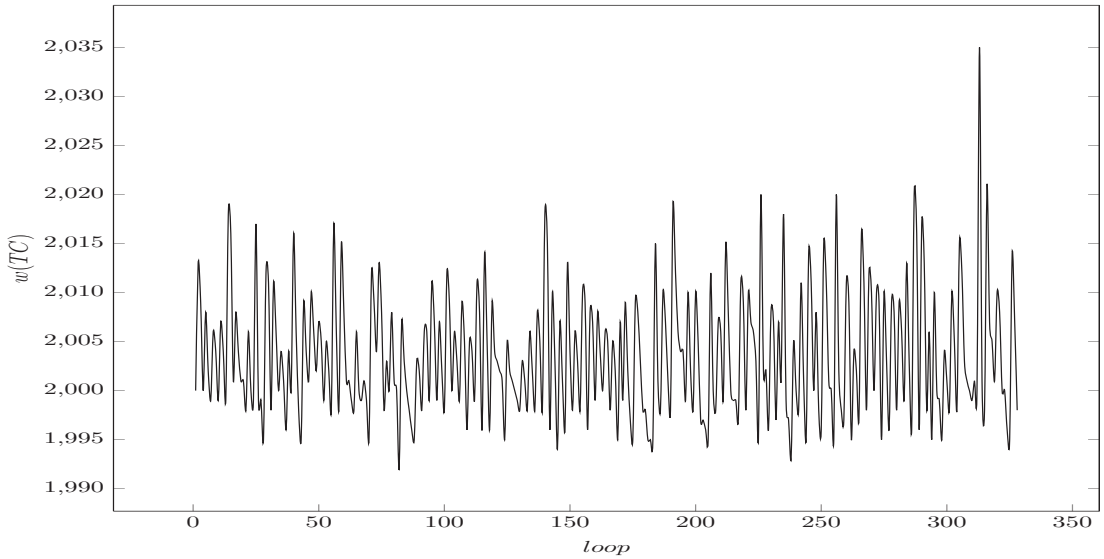


Figure 2.1: The picture illustrates the oscillation of the fitness function for instance $BPP\_766$ from `hard28` dataset. Total weight of non-assigned items, $w(TC)$, is represented after first and second variant of Tabu() procedure and after Descent().

decreasing order for example, or even the one for which FF would produce optimal solution for each instance.

Many experiments including choices of neighborhoods, termination conditions, etc. have been conducted. For instance, a partial solution with $m-3$ bins can be employed, and termination condition could be a packing of non-assigned items into three bins. This modification is capable of producing improved average results or computation times for certain instances, though our experiments have shown that no overall improvement can be achieved by applying it. On the other hand, one could simplify the procedure by adopting a partial solution with $m-1$ bins (or $m$ bins) and using $w(TC) \leq C$ (or $w(TC) = 0$) as the termination criteria. The first simplification will produce worse results only for `hard28` dataset, as will be shown later, while the second one yields significantly lower quality solutions on all datasets. This is quite understandable, given that, in this case, the only move to terminate is $Swap(0,1)$ (recall that no move between bins $\in B$ is performed) and a problem of obtaining a configuration for which this is possible is of same

difficulty as the original one (with capacity of one bin being decreased).

Since tabu search procedure is the main part of the algorithm, one can expect that the maximum number of items to rearrange in a single $Swap(p, q)$ move, i.e. the upper bounds for $p$ and $q$, can significantly influence the quality of results. As will be shown later, the results obtained by allowing no more than two items from the same bin to be swapped rather than three are only slightly worse. Nevertheless, allowing just $Swap(0, 1)$ and $Swap(1, 1)$ moves, drastically reduces solution quality. This outcome is to be expected since capacity violation is prohibited and most moves quickly become infeasible.

Only several parameters have been used during the entire method. Total running time of the algorithm has been limited to 60 seconds for all instances. Opting for a much smaller limit would produce the same results for most instances, but the 60-second limit is preferred mainly because of the difficulty of instances belonging to the `hard28` or `gau_1` class. Tabu Search procedure has been limited to a duration of $timeLimitTabu$, which has been set to one second in all reported experiments, and a maximum number of iterations (moves) without improvement, $maxNmbIters$, set to $|B| \times |I|$. Complete $Pack$ move is solely performed if the total number of items considered for rearrangement does not exceed 20 (which is not really a limitation for any of the instances considered herein) and a move limited to a subset of 20 items is performed otherwise. This limitation is introduced only to avoid huge enumeration times, but similar values (15-25 for example) will produce the same quality results. Furthermore, if dynamic programming procedure is utilized for finding an optimal packing, no limitation on number of items is required. Tabu tenure in tabu search procedure is proportional to the frequency of the move.

Optimizing running time was not the primary goal, so we did not exert much effort towards possibly accelerating the algorithm or finding solutions more quickly by making adaptations to it (e.g. exploring fewer neighborhoods) for certain instances. No distinction whatsoever has been made between the instances. Nevertheless, we have sought to obtain an algorithm with a reasonable running time, and we believe this goal has been achieved.

## 2.5 Applying the method on 2-DVPP and BPPC

The generalization of the presented method to 2-DVPP is straightforward. The main modification introduced is to the fitness function. Let $w^1(TC)$ and $w^2(TC)$ be the total weight of non-assigned items on resources one and two, respectively. The first objective here is to minimize the greater of the two values, $w^1(TC)/C_1$ and $w^2(TC)/C_2$. The second objective, like for BPP, is to maximize the number of items in the trash can. Formally, the following fitness function is to be minimized:

$$n \times C_1 \times C_2 \times max(\frac{w^1(TC)}{C_1}, \frac{w^2(TC)}{C_2}) - |TC| \tag{2.8}$$

As for BPP problem, upper bound $UB$ is obtained by First Fit heuristic applied on a randomly sorted set of items. Reduction procedure is analogous to the one used in BPP: a set of item pairs $(i, j)$ s.t. $w_i^1 + w_j^1 = C_1$ and $w_i^2 + w_j^2 = C_2$ has been identified, and the problem is reduced by fixing assignments of these items. In the case of 2-DVPP, we consider the weight of item $j$ to be greater than or equal to half of the bin capacity if $w_j^1 \geq \frac{C_1}{2}$ and $w_j^2 \geq \frac{C_2}{2}$.
Only a simple lower bound,

$$\left\lceil max(\frac{\sum_{i=1}^{n} w_i^1}{C_1}, \frac{\sum_{i=1}^{n} w_i^2}{C_2}) \right\rceil$$

has been used for 2-DVPP. All other algorithm features, such as parameters choice, remain the same.

An analogous adaptation could be made for VPP with more than two resources; however, experiments were not conducted since only a few relevant experimental results have been reported in the literature and most algorithms proposed have only been tested on 2-DVPP benchmarks.

BPPC problem is simply transformed to 2-DVPP and solved as described above. Transformation consists of setting the second resource capacity $C_2$ to the upper bound on number of items per bin, $k$, and weight of each item on the second resource to 1.

## 2.6 Computational results

This section will report the results of extensive computational experiments performed using the presented method on a broad set of test problems. Our method has been implemented in C++ and compiled using gcc 4.7.2 compiler in Ubuntu 14.04. All tests were run on a computer with an Intel Core i7-3770 CPU 3.40 GHz processor. All computation times are reported in seconds; if not otherwise specified, reported values always correspond to the total running time of the algorithm i.e. from the lower bound calculation and initial solution construction to the search termination.

### 2.6.1 BPP

A common set of one-dimensional bin-packing instances has been used to test the method; this set consists of five classes of instances: 1) a class developed by Falkenauer [1996] consisting of two sets, `uniform` and `triplets` (denoted respectively by `U` and `T` in the result tables), with each containing 80 instances; 2) a class developed by Scholl et al. [1997] consisting of three sets `set_1`, `set_2`, and `set_3`, containing 720, 480 and 10 instances respectively; 3) a class of instances developed by Schwerin and Wäscher [1999] containing two sets, `was_1` and `was_2`, with each set comprising 100 instances; 4) a class of instances developed by Wäscher and Gau [1996], called `gau_1`, containing 17 problem instances; 5) the `hard`28 class, consisting of the 28 difficult problem instances used, e.g. in Belov and Scheithauer [2006]. All instances can be downloaded from the Web page of the EURO Special Interest Group on Cutting and Packing (ESICUP) (ESICUP [2013]). Optimal solutions for all instances are known.

Optimal solutions for all instances in the first three classes (1570 instances in all) have been obtained by several heuristics, including HI_BP and GGA-CGT. HI_BP optimally solves 12 of the 17 instances in `gau_1`, while other recent heuristics yielded more optimal results (e.g. 15 by C_BP and 16 by Perturbation-SAWMBS and GGA-CGT). The only instance from this class that could not be solved optimally by any heuristic algorithm is "TEST0014".

Fleszar and Charalambous [2011] reported that their Perturbation-SAWMBS method could not solve to optimality more instances in the `hard28` dataset than the First Fit Decrease (FFD) procedure (5 out of the 28), even when drastically increasing the maximum number of iterations in their algorithm. The same applies to the HI_BP algorithm, as reported in Quiroz-Castellanos et al. [2015]. Most of the other proposed heuristics for the bin packing problem, including the best performers, cannot optimally solve more than 5 instances from this class. Nevertheless, a recently proposed genetic algorithm GGA-CGT (Quiroz-Castellanos et al. [2015]) finds optimal solutions to 16 instances. These authors also reported that more instances can be solved by increasing the population size (up to 22 instances when the population is increased from 500 to 10,000,000).

The exact methods based on Arc-Flow Formulation, as presented in (Brandão and Pedroso [2013]), can solve all instances to optimality within a reasonable computing time, including all instances from the `hard28` dataset. Solver can be downloaded at `http://vpsolver.dcc.fc.up.pt/` and detailed results obtained by using Gurobi solver for solving their Arc-Flow model are given at `http://www.dcc.fc.up.pt/~fdabrandao/research/vpsolver/results/`.
Average running times of this exact algorithm for each class of instances are listed in column *time* in Table 2.1. Computer used is 2 × Quad-Core Intel Xeon at 2.66GHz, Mac OS X 10.8.0, 16 GBytes of memory, while Gurobi 5.0.0 solver (single thread) is used to solve the model. It can be noted that the computation times in their experiments were much greater for `gau_1` dataset (up to a few thousand seconds) when compared to other datasets. This is due to the fact that average number of items per bin in this dataset is greater than in other datasets. Generally, one should not forget that exact algorithm runs until optimality is proven, but might find optimal solution very early and exact algorithm can be, for example, easily transformed into a heuristic approach by stopping it after a given time limit. However, this is not the case here since lower bounds, obtained by linear programming (LP) relaxation or by other well known methods, are equal to the optimal solutions in majority of the cases, and, thus, stopping the algorithm before optimality is proven will rarely produce optimal solutions. More precisely, lower bounds obtained by LP relaxation of Arc-Flow model are not equal to the

optimal solutions only for 7 out of 1615 instances considered here. Average running time elapsed from starting integer optimization (excluding bound calculation time) until finding the best solution for an exact approach is reported in column *IPtimeToBest* in Table 2.1. Table 2.2 reports total number of optimal solutions found by exact approach with limiting the computational time to different values and excluding time spent for calculating the bound.

| class | inst | time(s) | IPtimeToBest(s) |
|---|---|---|---|
| U | 80 | 0.34 | 0.24 |
| T | 80 | 0.91 | 0.71 |
| set_1 | 720 | 0.15 | 0.10 |
| set_2 | 480 | 43.4 | 39.7 |
| set_3 | 10 | 12.1 | 7.53 |
| was_1 | 100 | 0.67 | 0.52 |
| was_2 | 100 | 0.57 | 0.40 |
| gau_1 | 17 | 1641 | 1485 |
| hard28 | 28 | 29.69 | 27.0 |

Table 2.1: Results of exact approach based on Arc-Flow formulation

| Turning exact approach into heuristic | | | | | |
|---|---|---|---|---|---|
| timeLimit(s) | 60 | 120 | 300 | 600 | 1000 |
| opt solutions | 1520 | 1558 | 1598 | 1607 | 1611 |

Table 2.2: Number of optimally solved instances when limiting the running time of the exact approach based on Arc-Flow formulation.

To investigate the effectiveness of CNS_BP, we compared these results with those obtained by the best heuristic approaches reported in the literature, Perturbation-SAWMBS and GGA-CGT.
Running time of the algorithm is highly influenced by several important factors, such as a lower bound used to terminate the search and running time and iterations limit placed on the algorithm. Using a more complicated bound could terminate the search earlier but can also consume a significant CPU time (especially if implementation is suboptimal). Choosing the running time and iterations

limits largely depends on the set of instances to be solved. As an example, if the hard28 dataset was excluded from consideration, which is the case in many published papers on BPP, then the average total time would decrease substantially, as would the required running time and iterations limits. In the reported results, there is no distinction made in the algorithm across all instances. Raising the running time or iterations limit could increase the number of optimal solutions but might also drastically increase the average running time since total running time is to be reported, even if no improvement in results is achieved. We will also report the running times required to obtain the best solution values i.e. excluding the time spent exploring the search space after the last complete solution has been found. All results reported in this section have been obtained with a running time limit of 60 seconds and no more than 100,000 moves resulting in $w(TC) \leq 2C$, if not otherwise specified.

| Results for seed = 1 | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | | P.-SAWMBS | | | GGA-CGT | | CNS_BP | | | |
| class | inst | opt | time | scTime | opt | time | opt | time | scTime | toBest |
| U | 80 | 79 | 0.00 | 0.00 | 80 | 0.23 | 80 | 0.072 | 0.315 | 0.062 |
| T | 80 | 80 | 0.00 | 0.00 | 80 | 0.41 | 80 | 0.016 | 0.070 | 0.016 |
| set_1 | 720 | 720 | 0.01 | 0.015 | 720 | 0.35 | 720 | 0.083 | 0.364 | 0.002 |
| set_2 | 480 | 480 | 0.00 | 0.00 | 480 | 0.12 | 480 | 0.029 | 0.127 | 0.029 |
| set_3 | 10 | 10 | 0.16 | 0.24 | 10 | 1.99 | 10 | 0.001 | 0.004 | 0.001 |
| was_1 | 100 | 100 | 0.00 | 0.00 | 100 | 0.00 | 100 | 0.000 | 0.001 | 0.000 |
| was_2 | 100 | 100 | 0.01 | 0.015 | 100 | 1.07 | 100 | 0.000 | 0.001 | 0.000 |
| gau_1 | 17 | 16 | 0.04 | 0.06 | 16 | 0.27 | 17 | 3.131 | 13.69 | 2.352 |
| | | | | | | | * | 0.818 | 3.575 | 0.04 |
| hard28 | 28 | 5 | 0.24 | 0.36 | 16 | 2.40 | 25 | 6.205 | 27.14 | 3.351 |
| TOTAL | 1615 | 1590 | | | 1602 | | 1612 | | | |
| | | Intel core2 Q8200 2.33GHz | | | Core2 Duo CE6300 1.86GHz | | Intel i7-3770 3.40GHz | | | |

Table 2.3: Results with a seed set equal to 1. A comparison is drawn with the best state-of-the-art methods. The reported running time for gau_1 has been largely influenced by the "TEST0014" instance, which is not solved in any other heuristic; therefore, the running time without this instance has been reported as well(*).

Much like most of the previous work on BPP, for each instance, a single execution of the algorithm was run, with the initial seed for the random number generation set to 1. Results are listed in Table 2.3. For each class of instances,

| | 60 sec | | 30 sec | | 10 sec | | 5 sec | | 2 sec | |
|---|---|---|---|---|---|---|---|---|---|---|
| class | opt | time | opt | time | opt | time | opt | time | opt | time |
| U | 80.0 | 0.070 | 80.0 | 0.070 | 80.0 | 0.070 | 80.0 | 0.070 | 80.0 | 0.070 |
| T | 80.0 | 0.016 | 80.0 | 0.016 | 80.0 | 0.016 | 80.0 | 0.016 | 80.0 | 0.016 |
| set_1 | 719.6 | 0.074 | 719.6 | 0.074 | 719.6 | 0.074 | 719.6 | 0.070 | 719.6 | 0.054 |
| set_2 | 480.0 | 0.043 | 480.0 | 0.043 | 479.70 | 0.039 | 479.46 | 0.035 | 479.02 | 0.031 |
| set_3 | 10.0 | 0.001 | 10.0 | 0.001 | 10.0 | 0.001 | 10.0 | 0.001 | 10.0 | 0.001 |
| was_1 | 100.0 | 0.000 | 100.0 | 0.000 | 100.0 | 0.000 | 100.0 | 0.000 | 100.0 | 0.000 |
| was_2 | 100.0 | 0.001 | 100.0 | 0.001 | 100.0 | 0.001 | 100.0 | 0.001 | 100.0 | 0.001 |
| gau_1 | 16.90 | 2.371 | 16.70 | 1.987 | 16.24 | 1.366 | 16.12 | 0.896 | 16.0 | 0.390 |
| | * | - | | - | | - | | - | | 0.048 |
| hard28 | 24.78 | 7.224 | 24.52 | 4.750 | 23.64 | 2.672 | 22.96 | 1.763 | 21.34 | 0.958 |
| TOTAL | 1611.28 | | 1610.82 | | 1609.18 | | 1608.14 | | 1605.96 | |

Table 2.4: Average Results of CNS_BP for 50 seeds with different time limits

the number of optimally solved instances using each approach, as well as the average computation time per instance, are reported. For our approach, average time to obtain the best result ($toBest$) per instance for each class have also been reported. Algorithms are executed on different machines and to have a fair comparison we will calculate the scale factors, which are used to compare the performance of the algorithms as if they were running on the same machine. For this purpose, we will use the CPU speed estimations provided in SPEC standard benchmark ($https://www.spec.org/cpu2006/results/cint2006.html$). According to this, CPU speeds of processors used to run Perturbation-SAWMBS, GGA-CGT, and CNS_BP are 18.30, 12.30 and 53.80 respectively. If we choose the second one as a reference one, the scale factors will be 1.48, 1 and 4.37 respectively. Therefore, reported running times of each three algorithms are multiplied by these factors and reported in columns $scTime$. Note that this column is not necessary for GGA-CGT algorithm since corresponding factor is equal to 1.

A more detailed and more relevant (in our opinion) result of CNS_BP would be the average from running the algorithm for 50 different seeds (1 to 50) and for different time limits. These results are reported in Table 2.4. As in the previous table, the number of optimally solved instances and average running time per instance are indicated for each class and each of five time limits (60, 30, 10, 5 and 2 seconds).

Let's note that in terms of number of optimal solutions found, our algorithm

outperforms all published heuristic algorithms on the last two datasets and moreover obtains the same (optimal) solutions on all other datasets. Perturbation-SAWMBS is superior to other two algorithms in terms of running time, while running times for the previous best algorithm in terms of number of optimal solutions found, GGA-CGT, and CNS_BP are comparable.

It can be noticed that all instances from 6 datasets are solved in each of the runs when the running time limit is set at 1 minute. The same finding holds for 16 instances from the `gau_1`, 24 instances from the `hard28` and 718 instances from the `set_1`.. This outcome demonstrates the robustness and precision of the presented algorithm. The number of solved instances from the `hard28` dataset varies from 24 to 27 for the various seeds and detailed results are illustrated in Table 2.5. The only non-solved instance in 50 runs is "BPP_13". Nevertheless, optimal solution for this instance can be found when increasing the time limit or running the algorithm for more seeds. In most published papers, including HI_BP, Perturbation-SAWMBS and GGA-CGT, algorithm robustness has been verified by executing the algorithm five times with different seeds of random numbers (8075 runs in all). The average results should indeed be considered as a reference for each algorithm. The proposed algorithm found the optimal solutions in 8057 runs when running with seeds 1-5, missing the optimal solution in just 18 cases (once for "TEST0014" from `gau_1`, 15 times for `hard28` and twice for `set_1` ); in contrast, HI_BP, Perturbation-SAWMBS and GGA-CGT fail to obtain optimal solutions in 144, 128 and 78 cases, respectively.

The results derived by certain method simplifications are reported in Table 2.6. More specifically, the results when prohibiting more than two items from the same bin to be rearranged in a single *Swap* move during the tabu search procedure, the results obtained without the *Descent*() procedure and the results with partial solution containing $m - 1$ bins (and termination condition $w(TC) \leq C$) are included. This table reports the average results over 50 seeds. One can notice that first two simplifications do not significantly impact the quality of the results, while the third simplification produces worse solutions only on `hard28` dataset.

| instance | LB | OPT | #opt | time | toBest |
|---|---|---|---|---|---|
| BPP_14 | 61 | 62 | 50 | 9.508 | 0.000 |
| BPP_832 | 60 | 60 | 50 | 0.674 | 0.674 |
| BPP_40 | 59 | 59 | 16 | 50.711 | 10.362 |
| BPP_360 | 62 | 62 | 50 | 0.063 | 0.063 |
| BPP_645 | 58 | 58 | 50 | 1.374 | 1.374 |
| BPP_742 | 64 | 64 | 50 | 0.077 | 0.077 |
| BPP_766 | 62 | 62 | 50 | 7.716 | 7.716 |
| BPP_60 | 63 | 63 | 8 | 9.403 | 0.157 |
| BPP_13 | 67 | 67 | 0 | 7.356 | 0.000 |
| BPP_195 | 64 | 64 | 50 | 0.038 | 0.038 |
| BPP_709 | 67 | 67 | 11 | 49.516 | 7.887 |
| BPP_785 | 68 | 68 | 50 | 0.251 | 0.251 |
| BPP_47 | 71 | 71 | 50 | 0.136 | 0.136 |
| BPP_181 | 72 | 72 | 50 | 1.348 | 1.348 |
| BPP_359 | 75 | 76 | 50 | 1.438 | 0.000 |
| BPP_485 | 71 | 71 | 50 | 0.622 | 0.622 |
| BPP_640 | 74 | 74 | 50 | 0.048 | 0.048 |
| BPP_716 | 76 | 77 | 50 | 1.104 | 0.000 |
| BPP_119 | 76 | 77 | 50 | 59.372 | 0.000 |
| BPP_144 | 73 | 73 | 50 | 0.565 | 0.565 |
| BPP_561 | 72 | 72 | 50 | 0.012 | 0.012 |
| BPP_781 | 71 | 71 | 50 | 0.017 | 0.017 |
| BPP_900 | 75 | 75 | 50 | 2.642 | 2.642 |
| BPP_175 | 83 | 84 | 50 | 3.881 | 0.001 |
| BPP_178 | 80 | 80 | 50 | 0.241 | 0.241 |
| BPP_419 | 80 | 80 | 50 | 5.856 | 5.856 |
| BPP_531 | 83 | 83 | 50 | 0.093 | 0.093 |
| BPP_814 | 81 | 81 | 50 | 0.032 | 0.032 |

Table 2.5: Detailed results for hard28 dataset. The number of runs resulting in an optimal solution (#opt) is reported for each instance, as are the average running time (*time*) and average time spent to obtain best solutions (*toBest*). 50 runs were conducted for each instance. One can note that average running time over all instances is largely influenced by the running time on most difficult instances and instances for which the lower bound is different than optimal value (BPP_119 for example).

### 2.6.2 2-DVPP

2-DVPP instances used to evaluate the performance of CNS_BP have been extracted from Caprara and Toth [2001]; Spieksma [1994] and moreover have been addressed in Monaci and Toth [2006]. A total of 10 different classes of instances are

| | | $Swap(p \leq 2, q \leq 2)$ | | no $Descent()$ | | $m-1$ bins | |
|---|---|---|---|---|---|---|---|
| class | inst | opt | time | opt | time | opt | time |
| U | 80 | 80.0 | 0.062 | 80.0 | 0.070 | 80.0 | 0.053 |
| T | 80 | 80.0 | 0.013 | 80.0 | 0.015 | 80.0 | 0.051 |
| set_1 | 720 | 719.6 | 0.051 | 719.54 | 0.062 | 719.08 | 0.081 |
| set_2 | 480 | 479.88 | 0.065 | 479.26 | 0.127 | 480.0 | 0.017 |
| set_3 | 10 | 10.0 | 0.106 | 10.0 | 0.001 | 10.0 | 0.015 |
| was_1 | 100 | 100.0 | 0.021 | 100.0 | 0.001 | 100.0 | 0.001 |
| was_2 | 100 | 100.0 | 0.025 | 99.84 | 0.006 | 100.0 | 0.001 |
| gau_1 | 17 | 16.0 | 2.968 | 16.90 | 1.585 | 16.50 | 2.275 |
| hard28 | 28 | 24.54 | 7.198 | 24.58 | 6.786 | 17.16 | 25.280 |
| TOTAL | 1615 | 1610.02 | | 1610.12 | | 1602.74 | |

Table 2.6: Results with simplifications, average results for 50 seeds. Results when allowing a maximum of two items from the same bin to be swapped in the tabu search procedure, results without applying the Descent procedure and results with partial solution with $m-1$ bins are reported.

presented. Each class is composed of 40 instances, broken down into 10 instances of four different sizes. The Class 10 instances have been generated by cutting the bin resources into triplets of objects, such that not a single capacity slack unit is found in these solutions. For this class therefore, the optimal solutions in most cases are known as a consequence of the instance generation process, but not a result of bin packing algorithms. Classes 2, 3, 4, 5 and 8 are known to be easily solvable by simple greedy heuristics (Monaci and Toth, 2006), hence we shall focus our experiments on the remaining classes, which yield a total of 200 instances.

The best results are obtained by the iterated local search (MS-ILS) heuristic method, as reported in Masson et al. [2013], and we will compare our results to theirs. Nevertheless, 330 out of 400 instances could still be solved by the exact approach proposed by Brandão and Pedroso [2013]; 60 out of 70 unsolved instances belong to classes 4 and 5 and can be easily solved optimally by non-exact approaches. Consequently, 10 instances belonging to class 9 and containing 200 items are the only ones whose optimum remains unknown. Our method obtains optimal solutions for all instances with known optima (390 out of 400) and the same best known values for the remaining 10 instances.

The results reported in Table 2.7 have been aggregated by problem class, i.e. for each class the cumulative number of bins of 10 instances is reported. A total of 50 runs with different seeds are performed for each instance. In turn, each column presents the problem size, problem class, simple lower bound, optimal value (if known), the best known upper bound obtained by a heuristic (in this case $MS-ILS$), average running time per instance for $MS-ILS$, average and best number of bins generated with our heuristic, and the average CPU time per instance and average CPU time required to obtain the best results. All best known upper bounds and optimal solutions obtained by previous algorithms have been found. The running time limit was set to 10 seconds, and no more than two items from the same bin can be rearranged in a single *Swap* move throughout the tabu search procedure. $MS-ILS$ results reported in Masson et al. [2013] have been obtained with running the local search with a time limit of 300 seconds on an Opteron 2.4 GHz with 4 GB of RAM memory running Linux OpenSuse 11.1.

### 2.6.3 BPPC

To evaluate the performance of CNS_BP on BPPC, we used the same BPPC instances as Brandao and Pedroso (2013) for evaluating their Arc-Flow model. Their Arc-Flow model could solve all the instances to optimality (`http://www.dcc.fc.up.pt/~fdabrandao/research/vpsolver/results/`). Namely, BPPC instances have been created by adding cardinality constraints to one-dimensional BPP instances. For each of 1615 instances considered in BPP, the instance with cardinality $k \in [2, k\_max]$ has been created, where $k\_max$ is chosen such that optimal solution value for BPPC with cardinality $k\_max$ equals to the optimal solution value for problem without cardinality constraint (BPP). In total, 5255 instances have been created. Obtained results are reported in Table 2.8. Total number of instances and average number of optimal solutions obtained by proposed algorithm have been reported for each dataset. As for BPP, we run the algorithm with 50 different seeds for each instance. Results of similar quality as for BPP have been obtained, with average number of optimally solved instances being 5252.38. Running time limit was set to 60 seconds.

| 2-DVPP results - 50 seeds | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | MS-ILS | | CNS_BP | | | |
| N | class | LB | OPT | best | time | avg | best | time | toBest |
| 25 | 1 | 69 | 69 | 69 | 12.7 | 69.0 | 69 | 0.000 | 0.000 |
| 25 | 6 | 99 | 101 | 101 | 21.3 | 101.0 | 101 | 2.000 | 0.000 |
| 25 | 7 | 95 | 96 | 96 | 18.6 | 96.0 | 96 | 1.000 | 0.000 |
| 25 | 9 | 63 | 73 | 73 | 20.3 | 73.0 | 73 | 10.000 | 0.000 |
| 24 | 10 | 80 | 80 | 80 | 11.1 | 80.0 | 80 | 0.000 | 0.000 |
| 50 | 1 | 135 | 135 | 135 | 72.3 | 135.0 | 135 | 0.000 | 0.000 |
| 50 | 6 | 213 | 215 | 215 | 68.6 | 215.0 | 215 | 2.000 | 0.000 |
| 50 | 7 | 196 | 197 | 197 | 88.0 | 196.0 | 196 | 1.000 | 0.000 |
| 50 | 9 | 135 | 145 | 145 | 199.2 | 145.0 | 145 | 10.000 | 0.000 |
| 51 | 10 | 170 | 170 | 170 | 68.9 | 170.0 | 170 | 0.000 | 0.000 |
| 100 | 1 | 255 | 255 | 257 | 294.5 | 255.0 | **255** | 0.034 | 0.034 |
| 100 | 6 | 405 | 410 | 410 | 300.0 | 410.0 | 410 | 5.001 | 0.001 |
| 100 | 7 | 398 | 402 | 402 | 285.9 | 402.0 | 402 | 4.004 | 0.004 |
| 100 | 9 | 257 | 267 | 267 | 300.0 | 267.0 | 267 | 10.000 | 0.000 |
| 99 | 10 | 330 | 330 | 330 | 232.4 | 330.0 | 330 | 0.007 | 0.007 |
| 200 | 1 | 503 | 503 | 503 | 300.0 | 503.0 | 503 | 0.011 | 0.011 |
| 200 | 6 | 803 | 811 | 811 | 300.0 | 811.0 | 811 | 8.002 | 0.002 |
| 200 | 7 | 799 | 801 | 802 | 300.0 | 801.0 | **801** | 2.000 | 0.000 |
| 200 | 9 | 503 | — | 513 | 300.0 | 513.0 | 513 | 10.011 | 0.011 |
| 201 | 10 | 670 | 670 | 678 | 300.0 | 670.1 | **670** | 0.913 | 0.784 |

Table 2.7: 2-DVPP Results. Improvements over previous solutions found by heuristics are shown in bold.

| class | instBPP | card | totalInst | #opt |
|---|---|---|---|---|
| U | 80 | 2-3 | 160 | 160.0 |
| T | 80 | 2-3 | 160 | 160.0 |
| set_1 | 720 | 2-4 | 1189 | 1188.60 |
| set_2 | 480 | 2-10 | 2529 | 2528.54 |
| set_3 | 10 | 2-4 | 30 | 30.0 |
| was_1 | 100 | 2-6 | 500 | 500.0 |
| was_2 | 100 | 2-6 | 500 | 500.0 |
| gau_1 | 17 | 2-18 | 131 | 130.96 |
| hard28 | 28 | 2-3 | 56 | 54.28 |
| total | 1615 | | 5255 | 5252.38 |

Table 2.8: BPPC results

## 2.7 Conclusion

A new local search-based algorithm has been proposed for the Bin Packing Problem. The main feature of this algorithm is to proceed by partial configurations of the search space. Items are assigned in $m$ bins while respecting the capacity constraint or else are not assigned at all. The single goal of this algorithm is to derive a set of non-assigned items that can be packed into two bins; hence, we have obtained a complete $m + 2$-bin feasible solution. To continue, computations have been divided into two repeated steps:

- the tabu search on partial feasible configurations. Only low cardinality swap moves between non-assigned items and bin are used. The fitness function is aimed at minimizing the sum of weights of non-assigned items while maximizing the number of non-assigned items;
- the Descent procedure, which performs local optimal packing between non-assigned items and a bin with the same fitness function. This step corresponds to generalized swap moves between a bin and non-assigned items.

Whenever the sum of weights of non-assigned items is less than or equal to twice the capacity of a bin, the attempt is made to pack all non-assigned items into two bins by using the very same packing procedure as in the Descent step of the algorithm. The complexity of this procedure is bounded by $\mathcal{O}(k \times C)$ (*using dynamic programming like for the simple* $0-1$ *knapsack problem*), where $k$ is equal to the number of items in one bin plus the number of non-assigned items or $\mathcal{O}(2^k)$ when using the enumeration algorithm.
This algorithm introduces very few parameters and outperforms all previous heuristic approaches on a wide range of BPP instances. Let's note in particular that it obtains better results than other heuristics on `hard`28 and `gau_`1 datasets. In considering the simplicity of the entire method, it was ultimately quite easy to adapt it to the Vector Packing Problem and solve the available benchmarks with a significantly better performance than other published approaches.

When taking into account that this whole process never considers any bin state metrology, these results might come as a surprise. The algorithm actually only focuses on the configuration of non-assigned items, which may be viewed as

a major restriction. Yet on the other hand, integrating bin characteristics (*like* $\sum_{b \in B}(C - w(b))^2$ *for instance*) into the fitness function, in order to guide swap moves between the bins, could significantly increase $CPU$ time, though maximizing this function has not produced any better results.

Some of the algorithm features that showed to be crucial in obtaining high quality solutions include (1) the size of a partial solution and termination criteria i.e. exploring partial solutions with $m - 2$ bins and terminating the search when all non-assigned items can be packed into two bins (thus, producing complete feasible solution with $m$ bins), (2) defining a suitable fitness function i.e lexicographic fitness function minimizing total weight of non-assigned items first and maximizing a number of non-assigned items second and (3) introducing second tabu search variant consisting of a subset of moves that do not decrease the second objective.

# Chapter 3

# Machine Reassignment Problem

Machine Reassignment Problem (MRP), a very large scale combinatorial optimization problem proposed by Google and posted at ROADEF/EURO Challenge 2012 compatition, is addressed in this chapter. Highly effective local search approach will be presented.

## 3.1 Introduction

Data centers have become a common and essential element in the functioning of many modern companies. The unprecedented growth of demand for data processing, storage and networking makes these data centers indispensable. The same growth causes the growth of data centers in their size and complexity. At the same time, the data centers are essential for the cooperation and interaction among individuals, businesses, communications, academia, and government systems worldwide. Almost every global company has several global data centers (a dozen all around the world for Google) while local data centers exist in practically every business building.

The operational cost is often one-third of all costs associated with a modern data center. A great deal of attention is being paid today to the optimal management of data centers to improve the overall efficiency regarding energy, water use and greenhouse gas reductions. Many of these strategies appeared to be holistic and one of the most important parts is the optimization and simplification of archi-

tecture, processes and maintenance favoring the modular reusability and ease of re-deployment. Optimal use of assets, such as computer resources (CPU, RAM, network bandwidth, etc.), and scheduling the computer processes is an important part of the whole process. The problem of efficient (re)scheduling of processes becomes critical when data centers are refreshed or moved from one location to another.

Google, the company with probably the most extensive practical experience with data centers, organized in 2009, and 2011 two Industry summits about data centers' efficiency. The Google research team formalized and proposed the Google Machine Reassignment problem as a subject of ROADEF/EURO Challenge 2012 (see ROA [2012]). The aim of the problem is to improve the usage of a set of machines. Initially, each process is assigned to a machine. In order to improve machine usage, processes can be moved from one machine to another. Possible moves are limited by constraints that address the compliance and the efficiency of improvements and assure the quality of service.

We propose herein a Noisy Local Search method (NLS) for solving Machine Reassignment problem. The method, in a round-robin manner, applies the set of predefined local moves to improve the solutions along with multiple starts and noising strategy to escape the local optima. The numerical evaluations demonstrate the remarkable performance of the proposed method on MRP instances (30 official instances divided in datasets A, B and X) with up to 50,000 processes.

The chapter is organized as follows. Problem statement is presented in Section 3.2 and related work is addressed in Section 3.3. Section 3.4 presents simple and efficient calculation of a lower bound. This lower bound is sufficient to assess the high quality of many solutions produced by the method. Principle neighborhoods of the local search are presented in the first part of Section 3.5. In the remaining part of this section, we present some advanced components of local search and how they are composed in a general solution method. In Section 3.6, we present the computational study conducted on 30 official instances provided by Google. The chapter is concluded with possible extensions of the work and refinement of the presented method.

## 3.2 Problem specification and notations

A detailed description of the problem is given in the ROADEF/EURO Challenge subject (ROA [2012]), and we summarize it here.
The aim of this challenge is to improve the usage of a set of machines. A machine has several resources as for example RAM and CPU, and runs processes which consume these resources. Initially each process is assigned to a machine. In order to improve machine usage, processes can be moved from one machine to another. Possible moves are limited by hard constraints, as for example resource capacity constraints, and have a cost. A solution to this problem is a new process-machine assignment which satisfies all hard constraints and minimizes a given objective cost.

### 3.2.1 Decision variables

Let $\mathcal{M}$ be the set of machines, and $\mathcal{P}$ the set of processes. A solution is an assignment of each process $p \in \mathcal{P}$ to one and only one machine $m \in \mathcal{M}$; this assignment is noted by the mapping $M(p) = m$ in this document. The original assignment of process $p$ is denoted $M_0(p)$. Note the original assignment is feasible, i.e. all hard constraints are satisfied. For instance, if $\mathcal{M} = \{m_1, m_2\}$ and $\mathcal{P} = \{p_1, p_2, p_3\}$, then $M(p_1) = m_1, M(p_2) = m_1, M(p_3) = m_2$ means processes $p_1$ and $p_2$ run on machine $m_1$ and process $p_3$ runs on machine $m_2$.

### 3.2.2 Hard constraints

#### 3.2.2.1 Capacity constraints

Let $\mathcal{R}$ be the set of resources which is common to all the machines, $C(m, r)$ the capacity of resource $r \in \mathcal{R}$ for machine $m \in \mathcal{M}$ and $R(p, r)$ the requirement of resource $r \in \mathcal{R}$ for process $p \in \mathcal{P}$. Then, given an assignment $M$, the usage $U$ of a machine $m$ for a resource $r$ is defined as:

$$U(m, r) = \sum_{p \in \mathcal{P}, M(p) = m} R(p, r)$$

A process can run on a machine if and only if the machine has enough available capacity on every resource. More formally, a feasible assignment must satisfy the capacity constraints:

$$\forall m \in \mathcal{M}, \forall r \in \mathcal{R},\ U(m,r) \leq C(m,r)$$

#### 3.2.2.2 Conflict constraints

Processes are partitioned into services. Let $\mathcal{S}$ be a set of services. A service $s \in \mathcal{S}$ is a set of processes which must run on distinct machines. Note that all services are disjoint.

$$\forall s \in \mathcal{S}, (p_i, p_j) \in \mathcal{S}^2, p_i \neq p_j \rightarrow M(p_i) \neq M(p_j)$$

#### 3.2.2.3 Spread constraints

Let $\mathcal{L}$ be the set of locations, a location $l \in \mathcal{L}$ being a set of machines. Note that locations are disjoint sets. For each $s \in \mathcal{S}$ let $spreadMin(s) \in N$ be the minimum number of distinct locations where at least one process of service $s$ should run. The constraints are defined by:

$$\forall s \in \mathcal{S}, \sum_{l \in \mathcal{L}} min(1, |\{p \in s : M(p) \in \mathcal{L}\}|) \geq spreadMin(s)$$

#### 3.2.2.4 Dependency constraints

Let $\mathcal{N}$ be the set of neighborhoods, a neighborhood $n \in \mathcal{N}$ being a set of machines. Note that neighborhoods are disjoint sets. If service $s^a$ depends on service $s^b$, then each process of $s^a$ should run in the neighborhood of a $s^b$ process:

$$\forall p^a \in s^a, \exists p^b \in s^b \ and\ n \in \mathcal{N}\ such\ that\ M(p^a) \in n\ and\ M(p^b) \in n.$$

Note that dependency constraints are not symmetric, i.e. service $s^a$ depends on service $s^b$ is not equivalent to service $s^b$ depends on service $s^a$.

#### 3.2.2.5 Transient usage constraints

When a process $p$ is moved from one machine $m$ to another machine $m'$ some resources are consumed twice; for example disk space is not available on machine $m$ during a copy from machine $m$ to $m'$, and $m'$ should obviously have enough available disk space for the copy. Let $\mathcal{TR} \subset \mathcal{R}$ be the subset of resources which need transient usage, i.e. require capacity on both original assignment $M_0(p)$ and current assignment $M(p)$. Then the transient usage constraints are:

$$\forall m \in \mathcal{M}, \forall r \in \mathcal{TR}, \sum_{p \in \mathcal{P}, M_0(p)=m \vee M(p)=m} R(p,r) \leq C(m,r)$$

Note there is no time dimension in this problem, i.e. all moves are assumed to be done at the exact same time. Then for resources in $\mathcal{TR}$ this constraint subsumes the capacity constraint.

### 3.2.3 Objectives

The aim is to improve the usage of a set of machines. To do so a total objective cost is built by combining a load cost, a balance cost and several move costs.

#### 3.2.3.1 Load cost

Let $SC(m,r)$ be the safety capacity of a resource $r \in \mathcal{R}$ on a machine $m \in \mathcal{M}$. The load cost is defined per resource and corresponds to the used capacity above the safety capacity; more formally:

$$loadCost(r) = \sum_{m \in \mathcal{M}} max(0, U(m,r) - SC(m,r)).$$

#### 3.2.3.2 Balance cost

As having available CPU resource without having available RAM resource is useless for future assignments, one objective of this problem is to balance available resources. The idea is to achieve a given target on the available ratio of two different resources. Let $\mathcal{B}$ be a set of triples defined in $N \times \mathcal{R}^2$. For a given triple

$b = (r_1, r_2, target) \in \mathcal{B}$, the balance cost is:

$$balanceCost(b) = \sum_{m \in \mathcal{M}} max(0, target \times A(m, r_1) - A(m, r_2))$$

with $A(m, r) = C(m, r) - U(m, r)$. The total balance cost is the sum over all given triples.

#### 3.2.3.3 Process move cost

Some processes are painful to move; to model this soft constraint a process move cost is defined. Let $PMC(p)$ be the cost of moving the process $p$ from its original machine $M_0(p)$. Total process move cost is defined as:

$$processMoveCost = \sum_{p \in \mathcal{P}, M(p) \neq M_0(p)} PMC(p)$$

#### 3.2.3.4 Service move cost

To balance moves among services, a service move cost is defined as the maximum number of moved processes over services. More formally:

$$serviceMoveCost = \max_{s \in \mathcal{S}}(|\{p \in s : M(p) \neq M_0(p)\}|)$$

#### 3.2.3.5 Machine move cost

Let $MMC(m_{source}, m_{destination})$ be the cost of moving any process $p$ from machine $m_{source}$ to machine $m_{destination}$. Obviously for any machine $m \in \mathcal{M}$, $MMC(m, m) = 0$. The machine move cost is then the sum of all moves weighted by relevant $MMC$:

$$machineMoveCost = \sum_{p \in \mathcal{P}} MMC(M_0(p), M(p))$$

#### 3.2.3.6 Total objective cost

The total objective cost is a weighted sum of all previous costs. It is the cost to minimize.

$$
\begin{aligned}
totalCost = & \sum_{r \in \mathcal{R}} weight_{loadCost}(r) \times loadCost(r) \\
& + \sum_{b \in \mathcal{B}} weight_{balanceCost}(b) \times balanceCost(b) \\
& + weight_{processMoveCost} \times processMoveCost \\
& + weight_{serviceMoveCost} \times serviceMoveCost \\
& + weight_{machineMoveCost} \times machineMoveCost
\end{aligned}
\tag{3.1}
$$

### 3.2.4 Instances

The method has been tested on the official set of competition instances provided by Google and used for competitors algorithms evaluation. It consists of three sets of instances, A, B and X, each containing ten instances. The first dataset, A, has been available since the beginning of the competition and was used for qualification stage evaluation. It is a medium-size dataset containing instances with up to 1000 processes and 100 machines. Datasets B and X are larger datasets, containing instances with up to 50,000 processes and 5,000 machines, and have been used in the final stage of the competition. Set B has been available since the beginning of the final stage, while dataset X is a hidden set of instances used to test the robustness of algorithms (to prevent an over-fitting of the presented solution approaches to the known problem instances) and has become publicly available after the end of the competition. Datasets B and X have been in a very similar way and, therefore, for the sake of simplicity, some experimental results in this chapter might be given only for dataset B, while very similar results are obtained on X.

Basic characteristics of all three datasets are given in Tables 3.1 and 3.2. The tables show the number of processes ($|\mathcal{P}|$), the number of machines ($|\mathcal{M}|$), the number of resources ($|\mathcal{R}|$), the number of transient resources ($|\mathcal{TR}|$), the number of services ($|\mathcal{S}|$), the number of locations ($|\mathcal{L}|$), the number of neighborhoods ($|\mathcal{N}|$), the number of dependencies (*dep*), and the number of balance costs ($|\mathcal{B}|$) for each instance.

| Instances A | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Inst | $\|\mathcal{M}\|$ | $\|\mathcal{R}\|$ | $\|\mathcal{TR}\|$ | $\|\mathcal{P}\|$ | $\|\mathcal{S}\|$ | $\|\mathcal{N}\|$ | $\|\mathcal{L}\|$ | dep | $\|\mathcal{B}\|$ |
| a1_1 | 4 | 2 | 0 | 100 | 10 | 1 | 4 | 10 | 1 |
| a1_2 | 100 | 4 | 1 | 1000 | 10 | 2 | 4 | 10 | 0 |
| a1_3 | 100 | 3 | 1 | 1000 | 100 | 5 | 25 | 10 | 0 |
| a1_4 | 50 | 3 | 1 | 1000 | 10 | 2 | 4 | 10 | 1 |
| a1_5 | 12 | 4 | 1 | 1000 | 10 | 2 | 4 | 10 | 1 |
| a2_1 | 100 | 3 | 0 | 1000 | 0 | 1 | 1 | 0 | 0 |
| a2_2 | 100 | 12 | 4 | 1000 | 100 | 5 | 25 | 0 | 0 |
| a2_3 | 100 | 12 | 4 | 1000 | 125 | 5 | 25 | 10 | 0 |
| a2_4 | 50 | 12 | 0 | 1000 | 125 | 5 | 25 | 10 | 1 |
| a2_5 | 50 | 12 | 0 | 1000 | 125 | 5 | 25 | 10 | 0 |

Table 3.1: The table shows the characteristics of dataset A.

## 3.3 Related work

Despite its importance in data center applications, the specific characteristics of the MRP have not been adequately and extensively addressed in the literature before the problem has been proposed at ROADEF/EURO Challenge 2012 competition. Recently, during and after the competition, few papers have been published on this topic.

Mehta et al. [2012] (second placed team in the competition) proposed a Constraint Programming (CP) approach to solve a problem and obtained high quality results. Authors developed a CP formulation of the problem that is especially suited for a large neighborhood search approach (LNS). LNS approach consists of repeatedly selecting and solving subproblems. Subproblem selection is based on selecting a subset of the machines, and allowing all processes on those machines to be reassigned. Subproblem creation and solution updating have been done efficiently, which was crutial in obtaining high quality solutions in a limited computational time (5 minutes time limit has been imposed in the competition). Authors also experimented with a mixed-integer programming (MIP) model for LNS. Both MIP and CP-based LNS approaches find similar solutions on a medium-size set of instances (dataset A, up to 1000 processes and 100 machines), while on larger instances where the number of processes and machines can be up to 50000

| Instances B,X | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Inst | $\|\mathcal{M}\|$ | $\|\mathcal{R}\|$ | $\|\mathcal{TR}\|$ | $\|\mathcal{P}\|$ | $\|\mathcal{S}\|$ | $\|\mathcal{N}\|$ | $\|\mathcal{L}\|$ | dep | $\|\mathcal{B}\|$ |
| B1,X1 | 100 | 12 | 4 | 5000 | 500 | 5 | 10 | 30 | 0 |
| B2,X2 | 100 | 12 | 0 | 5000 | 500 | 5 | 10 | 30 | 1 |
| B3,X3 | 100 | 6 | 2 | 20000 | 1000 | 5 | 10 | 50 | 0 |
| B4,X4 | 500 | 6 | 0 | 20000 | 1000 | 5 | 50 | 60 | 1 |
| B5,X5 | 100 | 6 | 2 | 40000 | 1000 | 5 | 50 | 60 | 1 |
| B6,X6 | 200 | 6 | 0 | 40000 | 1000 | 5 | 50 | 60 | 1 |
| B7,X7 | 4000 | 6 | 0 | 40000 | 1000 | 5 | 50 | 60 | 1 |
| B8,X8 | 100 | 3 | 1 | 50000 | 1000 | 5 | 10 | 50 | 0 |
| B9,X9 | 1000 | 3 | 0 | 50000 | 1000 | 5 | 100 | 60 | 1 |
| B10,X10 | 5000 | 3 | 0 | 50000 | 1000 | 5 | 100 | 70 | 1 |

Table 3.2: The table shows the characteristics of datasets B and X.

and 5000, CP-based LNS is superior in memory use and the quality of solutions that can be found in limited time.

Another LNS approach has been presented by Brandt et al. [2014]. Similarly to the previous paper, solution is iteratively improved by selecting a subset of processes to be considered for reassignment and the new assignments are evaluated by a constraint program.

Jaskowski et al. [2015] (third placed team in the competition) present a hybrid metaheuristic approach for solving the problem. The approach consists of a fast greedy hill climber and a large neighborhood search, which uses mixed integer programming to solve subproblems. Hill climbing algorithm is employed to quickly improve the initial solution and further improvements are performed by a MIP-based large neighborhood search. The hill climber explores only a single neighborhood, which includes $shift(p, m)$ move that reassigns process $p$ from its current machine $m_0$ to another machine $m_1$, and first improving move is always selected. The second phase consists of selecting subproblems (similar to Brandt et al. [2014]; Mehta et al. [2012]) and solving them by MIP (IBM CPLEX solver 12.5).

Portal et al. [2015] (ranked fourth in the competition) propose a heuristic based on simulated annealing. The search explores shift and swap neighborhoods. Despite the simplicity of the method when compared to the previously described

approaches, high quality results have been obtained on a considered set of instances.

Masson et al. [2013] propose a Multi-Start Iterated Local Search method (MS-ILS) for solving MRP and Two-Dimensional Vector Packing provlem (2-DVPP). MS-ILS relies on simple shift and swap neighborhoods as well as problem-tailored shaking and specialized restart procedures.

Most of the teams competing in ROADEF/EURO Challenge 2012 competition used a variant of Local Search method (Large Neighborhood Search, Simulated Annealing, Late Acceptance, Tabu Search,...). Some authors also use Mixed Integer Programs, efficient only for very small instances or to solve the subproblems or simply to compute the lower bounds.

MRP is similar to the Generalized Assignment Problem (GAP) with some specific, and often hard to satisfy, constraints. GAP is a classical combinatorial optimization problem and some of the local moves presented in this chapter (e.g. shift, swap) could be found in GAP related literature, for example in Yagiura et al. [2004] and Yagiura et al. [1998]. Given $n$ jobs $J = \{1, 2, \ldots, n\}$ and $m$ agents $I = \{1, 2, \ldots, m\}$, the goal is to determine a minimum cost assignment subject to assigning each job to exactly one agent and satisfying a resource constraint for each agent. Assigning job $j$ to agent $i$ incurs a cost of $c_{ij}$ and consumes an amount $a_{ij}$ of resource, whereas the total amount of the resource available at agent $i$ is $b_i$. An assignment is a mapping $\alpha : J \to I$, where $\alpha(j) = i$ means that job $j$ is assigned to agent $i$. Then the generalized assignment problem is formulated as follows:

$$\begin{aligned} &minimize \quad cost(\alpha) = \textstyle\sum_{j \in J} c_{\alpha(j),j} \\ &subject\ to \quad \textstyle\sum_{j \in J, \alpha(j)=i} a_{i,j} \leq b_i, \quad \forall i \in I \end{aligned}$$

Multi-Resource Generalized Assignment Problem (MRGAP) is a more complex version of GAP where jobs require more than one type of resources.

Some of the most successful approaches for solving MRGAP (GAP) are tabu-search algorithm by Yagiura et al. (Yagiura et al. [2004] and Yagiura et al. [1998]), large-scale variable neighborhood search (zana Mitrovic-Minic and Punnen [2009]) and tabu-search by Diaz and Fernandez (Diaz and Fernandez [1998]).

## 3.4 Lower Bound

In this section we present a simple lower bound calculation for a given problem. The existence and the quality of a lower bound is essential for all practical solution approaches. The bound is simple and easy to calculate, yet it is, for a great number of instances, only a fraction of a percent from the optimal value. The load cost and the balance cost are the two most important components of the objective function and the lower bound proposed here is equal to the sum of lower bounds for these components. This same lower bound procedure has also been proposed in several other papers, including Mehta et al. [2012], Jaskowski et al. [2015] and Brandt et al. [2014].

### 3.4.1 Load Cost Lower Bound

Solution load cost is given by

$$LC = \sum_{r \in R} weight_{loadCost}(r) \times loadCost(r),$$

where $weight_{loadCost}(r)$ is a given weight for load cost of resource $r$.
We have:

$$\begin{aligned} loadCost(r) &= \sum_{m \in M} max(0, U(m,r) - SC(m,r)) \\ &\geq \sum_{m \in M} (U(m,r) - SC(m,r)) \\ &= \sum_{m \in M} U(m,r) - \sum_{m \in M} SC(m,r) \\ &= U(r) - SC(r) \end{aligned}$$

where $U(r) = \sum_{m \in M} U(m,r)$ is the total requirement for resource $r$ and $SC(r) = \sum_{m \in M} SC(m,r)$ is the total safety capacity. The total load cost lower bound is then equal to

$$\sum_{r \in R} weight_{loadCost}(r) \times (U(r) - SC(r))$$

### 3.4.2 Balance Cost Lower Bound

In a similar way, we obtain a lower bound on the balance cost. The balance cost is given by

$$BC = \sum_{b \in B} weight_{balanceCost}(b) \times balanceCost(b),$$

where $weight_{balanceCost}(b)$ is a given weight for balance cost.
We have:

$$\begin{aligned} balanceCost(b) &= \sum_{m \in M} max(0, target \times A(m, r_1) - A(m, r_2)) \\ &\geq \sum_{m \in M} (target \times A(m, r_1) - A(m, r_2)) \\ &= \sum_{m \in M} target \times A(m, r_1) - \sum_{m \in M} A(m, r_2) \\ &= target \times \sum_{m \in M} A(m, r_1) - \sum_{m \in M} A(m, r_2) \\ &= target \times A(r_1) - A(r_2), \end{aligned}$$

where $A(m, r) = C(m, r) - U(m, r)$, $b = (r_1, r_2, target) \in B$ and $A(r) = \sum_{m \in M} A(m, r)$ is the total available amount of resource $r$. The total balance cost lower bound is then equal to

$$\sum_{b=(r_1, r_2, target) \in B} weight_{balanceCost}(b) \times (target \times A(r_1) - A(r_2)).$$

Lower bounds for datasets A and B are listed in tables 3.3 and 3.4. Apart from the lower bound presented here, lower bounds obtained by solving Linear Programming relaxation of the MIP model of the problem are reported for dataset A and taken from Masson et al. [2013]. Such lower bounds could not be calculated for datasets B and X due to a huge size of MIP models.

The last remark about lower bounds is that they are calculated as a sum over resources. This fact gives us an opportunity to easily estimate the quality of a solution with respect to a subset of resources and then intensify the optimization regarding only one or several resources.

| Instance | Load LB | Balance LB | Total LB | LP_LB |
|---|---|---|---|---|
| a1_1 | 31 011 730 | 13 294 660 | 44 306 390 | 44 306 501 |
| a1_2 | 777 530 730 | 0 | 777 530 730 | 777 531 000 |
| a1_3 | 583 005 700 | 0 | 583 005 700 | 583 005 715 |
| a1_4 | 0 | 242 387 530 | 242 387 530 | 242 406 000 |
| a1_5 | 602 301 710 | 125 276 580 | 727 578 290 | 727 578 000 |
| a2_1 | 0 | 0 | 0 | 126 |
| a2_2 | 13 590 090 | 0 | 13 590 090 | 537 253 000 |
| a2_3 | 521 441 700 | 0 | 521 441 700 | 1 031 400 000 |
| a2_4 | 1 450 548 890 | 229 673 490 | 1 680 222 380 | 1 680 230 000 |
| a2_5 | 307 035 180 | 0 | 307 035 180 | 307 403 000 |

Table 3.3: Lower Bounds - instances A

| Instance | Load LB | Balance LB | Total LB |
|---|---|---|---|
| B1 | 3 290 754 940 | 0 | 3 290 754 940 |
| B2 | 31 188 860 | 983 965 000 | 1 015 153 860 |
| B3 | 156 631 070 | 0 | 156 631 070 |
| B4 | 0 | 4 677 767 120 | 4 677 767 120 |
| B5 | 922 858 550 | 0 | 922 858 550 |
| B6 | 0 | 9 525 841 820 | 9 525 841 820 |
| B7 | 0 | 14 833 996 360 | 14 833 996 360 |
| B8 | 1 214 153 440 | 0 | 1 214 153 440 |
| B9 | 10 050 999 350 | 5 834 370 050 | 15 885 369 400 |
| B10 | 0 | 18 048 006 980 | 18 048 006 980 |

Table 3.4: Lower Bounds - instances B

## 3.5 Proposed Heuristic

This section describes the proposed Noisy Local Search method (NLS) approach used to solve the problem. The heuristic combines a Local-Search (LS) improvement procedure with the problem tailored noising moves, sorting the set of processes and restart procedures. The method iteratively tries to replace the current assignment with a better one. Only feasible moves are considered. Three different local search neighborhoods are explored. The initial solution, given as an input, is used as a starting point for the local search procedure. The search terminates once a maximum time limit is reached. We will first define all the neighborhoods

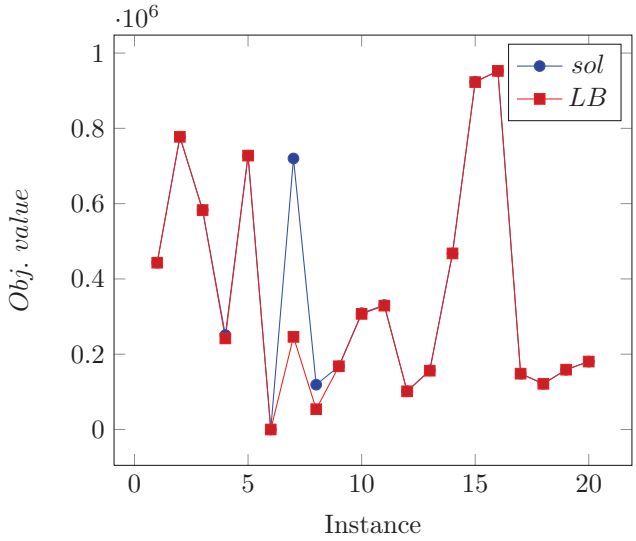


Figure 3.1: Gap between our best solutions and lower bounds on A and B datasets

explored in local search procedure and explain the evaluation procedures, followed by some strategies used to improve the solutions including intensification and diversification of the search.

As opposed to the presented local search method for Bin Packing for example, only moves that improve the objective function are accepted during the whole procedure. This is mainly due to the size of the problem at hand.

### 3.5.1 Neighborhoods

The local search procedure consists of exploring three neighborhoods, denoted by $shift$, $swap$ and $BPR$ and explained below. These three neighborhoods are explored sequentially, one after another, in a given order. The order of neighborhoods exploration in the final release of our method will be defined and discussed later. Exploring each of three neighborhoods terminates when no improvement moves are found or running time limit is reached.

#### 3.5.1.1 Shift and Swap

Given a solution $s$, the $shift$ neighborhood, $N_{shift}(s)$ , is defined to be the set of solutions that can be obtained from $s$ by reassigning one process from one machine to another. Formally,

$$N_{shift}(X) = \{X' : X' \text{ is obtained from X by changing the assignment of one process}\}$$

Given a solution $s$, the $swap$ neighborhood, $N_{swap}(s)$ , is the set of solutions that can be obtained from $s$ by interchanging the assignments of two processes assigned to different machines. Formally,

$$N_{swap}(X) = \{X' : X' \text{ is obtained from X by exchanging the assignments of two processes}\}$$

$Shift$ and $swap$ neighborhoods are the simplest ones and can be found in many papers concerning solving problems similar to MRP (Diaz and Fernandez [1998]; Yagiura et al. [1998, 2004]) and in several papers on MRP (Jaskowski et al. [2015]; Masson et al. [2013]; Portal et al. [2015]). Shift move that reassigns process $p \in \mathcal{P}$ from its current machine $M(p)$ to machine $m \in \mathcal{M}$ will be denoted by $shift(p, m)$, and exchanging the assignments of two processes $p_1, p_2 \in \mathcal{P}$ will be denoted by $swap(p_1, p_2)$. Change of the objective function with performing $shift(p, m)$ will be denoted by $eval\_shift(p, m)$. Obviously, the move is improving the objective function only if $eval\_shift(p, m) < 0$. In case $shift(p, m)$ move is not feasible (i.e. will result in constraints violation) we set $eval\_shift(p, m) = 1,000,000$. Analogous definition, $eval\_swap(p_1, p_2)$ holds for a swap move $swap(p_1, p_2)$.

As mentioned earlier, MRP is a very large scale problem with real-world benchmarks containing several thousands of processes (machines) and a huge number of constraints. For example, in a classic MIP formulation, instance B10 would have more than 250,000,000 binary decision variables. Furthermore, a running time limit of 5 minutes has been imposed in the competition. Consequently, even a complete evaluation of $shift$ and $swap$ neighborhoods is not possible. Obviously, complexity of $shift$ and $swap$ neighborhoods is $\mathcal{O}(|\mathcal{P}||\mathcal{M}|)$ and $\mathcal{O}(|\mathcal{P}|^2)$ respectively. Therefore, our local search procedure explores only a part of these neighborhoods as explained below. For the same reasons, choosing the "best" shift of swap move

in terms of the objective improvement each time (as was the case for example in proposed method for Bin Packing) is not reasonable.

Shift and swap neighborhoods are explored in a following way:

1. randomly select and evaluate a subset of possible moves $S$
2. sort the moves according to the objective improvement
3. repeat the following until $S$ is empty
    - scan the set $S$ and perform a move if it is still improving the objective or delete it from the list otherwise
4. repeat previous steps until stopping criteria is met

Note that performing one shift/swap move can change the value of another move in terms of the objective improvement of move feasibility. Therefore, in Step 2 of the previous procedure, the move is evaluated again and accepted only if objective function is improving. This change of moves quality also implies that, except for the first move performed, not the best move is always accepted. However, selecting the moves in a given way produces slightly better results than accepting the first improving move found or not sorting the list of moves, while, on the other hand, explores the neighborhood much faster than when evaluating all the moves after each performed move.

Selecting a subset of possible shift moves (step 1 in a previous procedure) is done in the following way:

- select a random subset of processes $P_1 \subseteq \mathcal{P}$. For each process $p \in P_1$ select a subset of machines $M_p \subseteq \mathcal{M}$ and evaluate all $shift(p, m)$ moves reassigning process $p$ to machine $m \in M_p$. Size of the set $P_1$ is limited to 10,000, while the size of $M_p$ is limited to 1,000.

Selecting possible swap moves is analogous:

- select a random subset of processes $P_1 \subseteq \mathcal{P}$. For each process $p_1 \in P_1$ select a subset of processes $P_2 \subseteq \mathcal{P}$ (no process in $P_2$ is currently assigned to the same machine as $p$) and evaluate all $swap(p_1, p_2)$ moves that exchange the assignments of processes $p_1$ an $p_2 \in P_p$. Size of both sets, $P_1$ and $P_2$ is limited to 2,500.

Pseudo codes for exploring shift and swap neighborhoods are given in Algorithms 3.1 and 3.3, while creating the list of possible moves is given in Algorithms 3.2 and 3.4.

**Algorithm 3.1:** shift()

```
   // Fill the list of all feasible and good shifts
1  S ← fillShift() ;
2  while S ≠ ∅ do
3      if first move shift(p0, m0) in S is feasible and eval_shift(p0, m0) < 0 then
4          perform move shift(p0, m0) - reassign p0 to m0 ;
5      else
6          Delete move shift(p0, m0) from the list S;
7      S ← fillShift() ;
```

**Algorithm 3.2:** fillShift()

```
   // set of moves
1  S ← ∅;
2  select a random subset of processes P1 ⊆ P of size min(10000, |P|);
3  for each process p ∈ P1 do
4      select a random subset of machines Mp ⊆ M of size min(1000, |M|);
5      for each machine m ∈ Mp do
6          if eval_shift(p, m) < 0 then
7              add shift(p, m) move to S;
8  return S;
```

For the set of instances provided in the ROADEF/EURO competition it has been observed that the original solution can be substantially improved by only exploring shift and swap neighborhoods. Furthermore, solutions obtained for majority of the instances from datasets B and X are near optimal. Results obtained by exploring only shift and swap neighborhoods for datasets A and B with a 5 minute running time limit are listed in table 3.5. Neighborhoods are explored in

**Algorithm 3.3:** swap()

```
   // Fill the list of all feasible and good shifts
1  S ← fillSwap() ;
2  while S ≠ ∅ do
3      if first move swap(p1, p2) in S is feasible and eval_swap(p1, p2) < 0 then
4          perform move swap(p1, p2);
5      else
6          Delete move swap(p1, p2) from the list S;
7      S ← fillSwap() ;
```

**Algorithm 3.4:** fillSwap()

```
   // set of moves
1  S ← ∅;
2  select a random subset of processes P1 ⊆ P of size min(2500, |P|);
3  for each process p1 ∈ P1 do
4      select a random subset of processes P2 ⊆ P of size min(2500, |P|);
5      for each process p2 ∈ P2 do
6          if eval_swap(p1, p2) < 0 then
7              add swap(p1, p2) to S;
8  return S;
```

turn until no improvement move can be found. One can note that shift and swap neighborhoods can be explored in a short time for dataset A, while for dataset B exploration usually does not terminate before a given time limit of five minutes has been reached.

| Instance | Initial | shift+swap | Lower Bound | cpu (s) |
|---|---|---|---|---|
| a1_1 | 49 528 750 | 44 306 501 | 44 306 390 | 0 |
| a1_2 | 1 061 649 570 | 803 075 488 | 777 530 730 | 0 |
| a1_3 | 583 662 270 | 583 006 315 | 583 005 700 | 0 |
| a1_4 | 632 499 600 | 278 499 816 | 242 387 530 | 1 |
| a1_5 | 782 189 690 | 727 579 209 | 727 578 290 | 1 |
| a2_1 | 391 189 190 | 4 545 591 | 0 | 1 |
| a2_2 | 1 876 768 120 | 993 897 289 | 13 590 090 | 1 |
| a2_3 | 2 272 487 840 | 1 454 003 869 | 521 441 700 | 0 |
| a2_4 | 3 223 516 130 | 1 797 566 378 | 1 680 222 380 | 5 |
| a2_5 | 787 355 300 | 491 424 615 | 307 035 180 | 3 |
| B1 | 7 644 173 180 | 3 684 274 572 | 3 290 754 940 | 30 |
| B2 | 5 181 493 830 | 1 025 573 054 | 1 015 153 860 | 148 |
| B3 | 6 336 834 660 | 157 975 099 | 156 631 070 | 300 |
| B4 | 9 209 576 380 | 4 677 901 406 | 4 677 767 120 | 300 |
| B5 | 12 426 813 010 | 923 321 429 | 922 858 550 | 300 |
| B6 | 12 749 861 240 | 9 525 919 157 | 9 525 841 820 | 136 |
| B7 | 37 946 875 350 | 14 932 536 292 | 14 833 996 360 | 300 |
| B8 | 14 068 207 250 | 1 214 435 602 | 1 214 153 440 | 288 |
| B9 | 23 234 641 520 | 15 885 653 135 | 15 885 369 400 | 300 |
| B10 | 42 220 868 760 | 18 171 973 406 | 18 048 006 980 | 300 |

Table 3.5: Shift+Swap results: neighborhoods are explored as described and exploration terminates when no improvement move is found. For each instance in datasets A and B, initial solution, improved solution by $shift$ and $swap$, lower bound and total exploration time (in seconds) are given. All the results are obtained using the same seed for a random number generator.

One can note that exploration of $shift$ and $swap$ neighborhoods described above is completely deterministic if number of processes does not exceed 2,500 and number of machines does not exceed 1,000 (this is the case for all dataset A instances). Results with slightly different way of exploring the neighborhoods are given in Table 3.6. Namely, instead of performing the first feasible move from set $S$ that improves the objective function, we randomly choose one of the first few

such moves (reported results are obtained by choosing one of first three moves). The search starts from initial solution and terminates when no improvement move can be found and this whole procedure repeats until time limit of 5 minutes is reached. The best solution found is reported. Results are reported only for several instances, i.e. those with the biggest gap from the lower bound in Table 3.5.

| Instance | Initial | shift+swap | Lower Bound |
|---|---|---|---|
| a1_2 | 1 061 649 570 | 785 827 517 | 777 530 730 |
| a1_4 | 632 499 600 | 267 891 485 | 242 387 530 |
| a2_1 | 391 189 190 | 4 513 190 | 0 |
| a2_2 | 1 876 768 120 | 986 680 470 | 13 590 090 |
| a2_3 | 2 272 487 840 | 1 430 489 492 | 521 441 700 |
| a2_4 | 3 223 516 130 | 1 746 529 151 | 1 680 222 380 |
| a2_5 | 787 355 300 | 468 937 777 | 307 035 180 |
| B1 | 7 644 173 180 | 3 588 464 895 | 3 290 754 940 |

Table 3.6: Shift+swap with randomness. Average results for 10 runs with different seeds are reported.

#### 3.5.1.2 Big Process Rearrangement (BPR) Neighborhood

The BPR neighborhood, $N_{bpr}(s)$, is the set of solutions $s'$ obtainable from $s$ by shifting a process $p$ to a machine $m$ while at the same time shifting a certain number of processes from $m$ to some other machines. A single BPR move will be denoted by BPR($p$). This neighborhood showed to be particularly useful in reassigning big processes. Often it is not possible to reassign a big process using shift and swap moves, especially if it is much bigger then all other processes. The BPR neighborhood can be very useful in a such situation. One should note that while performing this move, several processes from the same machine change their assignments, while shift and swap moves reassign only one process from or to a machine. Since the BPR neighborhood is more complex than shift and swap neighborhoods, first BPR move that improves the objective function is accepted.

The pseudo code for exploring the BPR neighborhood (a single BPR move) is given in Algorithm 3.5.
Process 'p' (line 1 in Algorithm 3.5) is chosen in the following way: All machines

**Algorithm 3.5:** BPR - one move

```
1 Choose process p (random process assigned to one of several most expensive machines);
2 for m = 1 to NM do
3     Reassign p to m;
4     Reassign processes from m (different from p) while cost is improving (only feasible
      reassignments);
5     if capacity violated on m then
6         Reassign processes from m until capacity satisfied (if possible);
7     if capacity violated or cost not improved then
8         Undo all the moves;
```

are sorted in descending order by the current machine cost. Machine cost is equal to the sum of current machine load cost and current machine balance cost, which are simple to calculate. Then, process selected for BPR move is a random process from one of several most expensive machines. Selected machine is also chosen randomly. Number of most expensive machines to be considered here is a parameter and it is set to $\frac{|M|}{20}$ in the experiments presented here. Both random choices, i.e. choice of process and choice of machine, are made using uniform distribution. One can note that, in this way, processes selected for BPR move are not necessarily "big processes", but we call the move "Big Process Rearrangement" move since it is especially useful in reassigning big processes.

During the whole BPR move, all constraints except capacity constraint (conflict, spread, dependency) are satisfied at any time. Capacity constraint can be possibly violated by reassigning process 'p' (line 1 in Algorithm 3.5). If possible, violated constraint will be satisfied by reassigning several other processes from the machine process $p$ was reassigned to. If all constraints are satisfied at the end of BPR move and solution cost has improved, BPR move is accepted. Otherwise, the next machine to reassign to or next BPR move are selected. Maximum number of machines selected (line 2 in Algorithm 3.5) is set to 100 in our experiments. Set of machines is also chosen randomly. As can be seen in Algorithm 3.5, examination of BPR neighborhood is brute force and straight forward and thus can be computationally expensive if used too much. Therefore, searching the BPR neighborhood is controlled by parameter $BPR$ defined in parameters section (Section 3.5.2.5). This parameter represents the maximum number of evaluated BPR moves (i.e. number

of calls to Algorithm 3.5).

Results obtained by exploring all three defined neighborhoods are listed in Table 3.7. As before, exploration of neighborhoods is done sequentially (order is $shift + swap + BPR$) and the best solution obtained in 5 minutes time frame are reported. Number of BPR moves evaluated is set to 300.

Table 3.8 illustrates the importance of the BPR neighborhood by comparing

| Instance | Initial | shift+swap+BPR | Lower Bound | cpu (s) |
|---|---|---|---|---|
| a1_1 | 49 528 750 | 44 306 501 | 44 306 390 | 300 |
| a1_2 | 1 061 649 570 | 782 975 028 | 777 530 730 | 300 |
| a1_3 | 583 662 270 | 583 006 015 | 583 005 700 | 300 |
| a1_4 | 632 499 600 | 275 698 786 | 242 387 530 | 300 |
| a1_5 | 782 189 690 | 727 578 509 | 727 578 290 | 300 |
| a2_1 | 391 189 190 | 314 | 0 | 300 |
| a2_2 | 1 876 768 120 | 911 613 753 | 13 590 090 | 300 |
| a2_3 | 2 272 487 840 | 1 431 708 958 | 521 441 700 | 300 |
| a2_4 | 3 223 516 130 | 1 684 411 144 | 1 680 222 380 | 300 |
| a2_5 | 787 355 300 | 390 622 267 | 307 035 180 | 300 |
| B1 | 7 644 173 180 | 3 475 699 536 | 3 290 754 940 | 300 |
| B2 | 5 181 493 830 | 1 023 723 645 | 1 015 153 860 | 300 |
| B3 | 6 336 834 660 | 157 460 196 | 156 631 070 | 300 |
| B4 | 9 209 576 380 | 4 677 920 811 | 4 677 767 120 | 300 |
| B5 | 12 426 813 010 | 923 766 440 | 922 858 550 | 300 |
| B6 | 12 749 861 240 | 9 525 941 777 | 9 525 841 820 | 300 |
| B7 | 37 946 875 350 | 14 835 299 922 | 14 833 996 360 | 300 |
| B8 | 14 068 207 250 | 1 214 569 926 | 1 214 153 440 | 300 |
| B9 | 23 234 641 520 | 15 885 704 020 | 15 885 369 400 | 300 |
| B10 | 42 220 868 760 | 18 048 531 587 | 18 048 006 980 | 300 |

Table 3.7: Shift+Swap+BPR results. Several solutions are built (until time limit of 5 minutes is reached) and the best one is reported. Average results for 10 runs with different seeds are reported.

the solutions obtained using all the neighborhoods to the solutions obtained using shift and swap neighborhoods only. The significant difference can only be seen on the most challenging instances (in our opinion $a2\_2$, $a2\_3$, $a2\_5$, $B1$ and $X1$). These instances are also the ones that had the largest impact on the results of

qualification and final phase of ROADEF/EURO competition, since the majority of teams obtain near optimal solutions on all other instances (especially on instances $B2-B10$, $X2-X10$). We can note that exploring the BPR neighborhood increases the running time of the method 3-7 times on these instances, but still can be comfortably used in the 5 minute time frame. Eventual speeding up the search through the BPR neighborhood could improve the method, and this could be the subject of the future work.

| | without $BPR$ | | without $ShiftSwap$ | |
|---|---|---|---|---|
| Inst | gap | speed up | gap | speed up |
| a1_2 | 0.33 | 4.1 | 0.08 | 1.19 |
| a1_4 | 0.90 | 4.3 | 0.30 | 1.21 |
| a2_2 | 13.1 | 6.8 | 0.45 | 1.12 |
| a2_3 | 7.5 | 6.6 | 0.61 | 1.15 |
| a2_5 | 8.2 | 3.1 | 0.77 | 1.22 |
| B1 | 0.5 | 3.2 | 0.05 | 1.07 |
| X1 | 0.4 | 3.2 | 0.07 | 1.10 |

Table 3.8: The importance of neighborhoods. The second column represents the gap between solutions obtained without the BPR neighborhood to the standard solutions (solutions using all neighborhoods). The third column represents the speed up of the method when not exploring the BPR neighborhood. Columns 4 and 5 illustrate the same thing for Shift+Swap neighborhood.

### 3.5.2 Tuning the algorithm

The core elements of the solution are already given and their straightforward or even naive use can solve to near optimality most of B and X instances, as showed in Table 3.7. Several other carefully designed and chosen implementation details render the solution robust and very efficient for all instances.

#### 3.5.2.1 Neighborhoods exploration

Local search neighborhoods are explored sequentially, meaning that the next neighborhood is explored after the previous has been finished. The order of neighborhoods exploration during the search showed to be important and the following

order has been used: BPR, shift, swap. As discussed earlier, shift and swap neighborhoods can be explored until no improvement move exists and all the results reported above have been obtained by using this termination condition. However, a large amount of CPU time can be consumed before stopping the search, especially for the biggest instances. As can be seen in Table 3.5, for some of the instances (B7, B10) search exploration will not terminate before 5 minute running time limit. On the other hand, as could be expected, most of the solution improvement has been done in first few loops (shift() + swap()), with most of the further improvement often being negligable. It can also happen that slow exploration of a single neighborhood (shift or swap) results in a small number of loops. Therefore, we decided to stop the exploration of neighborhood when total objective improvement between the last two calls to $fillShift()(fillSwap())$ is smaller than a given threshold, $minImprovement$. When stopping criteria is met for a current neighborhood, the search continues with exploring the next neighborhood.

#### 3.5.2.2 Sorting processes

The numerical experiments show that the quality of the solution is sensitive to the the order of processes to reassign. In the presented method we order the processes by their size and the reassignment of the large processes is done first followed by the reassignment of smaller processes.
The size of a process $p \in \mathcal{P}$ is defined as a sum of its requirements over all resources and denoted by $size(p)$. Formally speaking,

$$size(p) = \sum_{r \in \mathcal{R}} R(p, r).$$

The algorithm will first explore local search neighborhoods by considering only the rearrangements that include the biggest processes (first 10% for example). Then, the number of processes to consider for rearrangement is iteratively increased and exploring the search space continues.
More precisely, all the processes are sorted in decreasing order by $size(p)$ and the number of processes to consider is set to zero at the beginning of algorithm ($N = 0$). Then, in each iteration of the algorithm the number of processes to

consider is increased by $\delta$ and local search procedure considering (and reassigning) only processes from position 0 to $N$ in the sorted list of processes (denoted by $LS(N)$) is invoked. Obviously, $shift(p, m)$ can be performed in $LS(N)$ only if $p$ is one of $N$ biggest processes, while $swap(p_1, p_2)$ move if at least one of two processes, $p_1$ and $p_2$ is amongst $N$ biggest processes. Similarly, a necessary condition for a BPR($p$) move to be considered is related to the size of the first process (process $p$) to be shifted. This procedure is repeated until all processes are examined.
Sorting the processes by their size is only one of the possibilities, while the best sort remains one of the open questions in this research. Several experimentations have been done with defining the order of processes, but no improvement in the final results could be achieved. Value $\delta$ is a parameter and will be addressed in Section 3.5.2.5.

Results obtained when sorting the set of processes and limiting the search for the biggest processes only are listed in Table 3.9. Only the most difficult instances results are reported and one can note that a significant improvement has been achieved.

| Instance | shift+Swap+BPR+sorting |
|---|---|
| a1_2 | 780 559 151 |
| a1_4 | 270 997 812 |
| a2_2 | 799 925 251 |
| a2_3 | 1 287 389 046 |
| a2_4 | 1 684 612 901 |
| a2_5 | 346 851 450 |
| B1 | 3 434 975 791 |

Table 3.9: Results obtained with sorting the processes. Average results for 10 runs with different seeds and $\delta = |\mathcal{P}|/5$ are reported.

#### 3.5.2.3 Noising

The principles and details of the noising methods are thoroughly explained in Charon and Hurdy [2012]. The quality of the actual solution with respect to a given resource can be easily estimated using the calculation like the one explained in Section 3.4. The proposed noising method consists of increasing the load cost

weights for one or a subset of resources. This simple change in the objective function will diversify and intensify the search at the same time.
There are several possible ways to change the objective function. We implement and keep the most simple one by changing only one load cost weight at time:

- choose resource r
- increase load cost weight of the resource r.

We optimize the modified objective function and then continue, from obtained solution, with the optimization of the original objective. This is repeated for few different resources, which are chosen by importance (distance to the load cost lower bound). The weight increasing value used in presented results is equal to 10. The impact of sorting the processes and noising method to the final results are illustrated in Figure 3.2.

#### 3.5.2.4 Randomness - dealing with seeds and restarts

The proposed solution is still sensitive to the choice of the random seed and the pseudo-random function. The initial phase of the search, dealing with biggest processes, strongly influences the quality of the final solution. In other words, the final solution is usually good only if it is good after reassigning the biggest processes.
In order to increase the robustness of the method, the following approach is implemented:

- Optimize big processes reassignment (initial phase of the search) for several different seeds
- Use only a few best seeds in the remaining search

The exact number of different seeds to be tested depends mostly on the size and the type of the instance at hand. To be able to consider all the processes, optimizing big processes reassignment should not take too much CPU time and is limited in the proposed method to maximum 60 seconds. The maximum number of seeds is equal to 15 for set A instances and 8 for sets B and X.

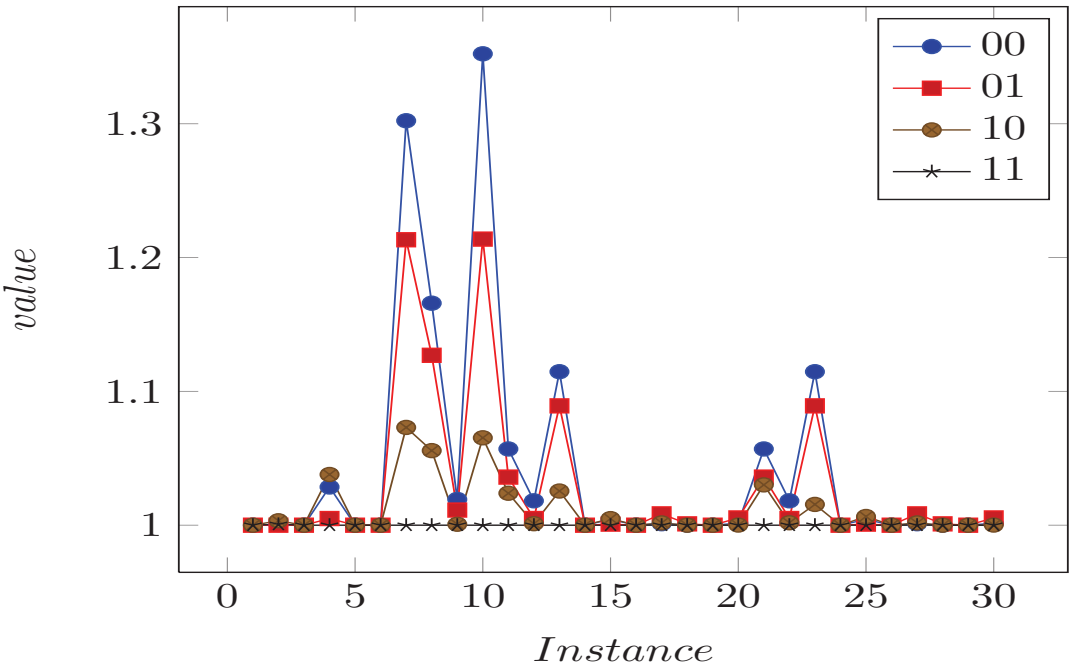


Figure 3.2: Results with/without sorting processes and with/without noising (00 - no sorting, no noising; 01 - no sorting; 10 - no noising; 11 - with sorting and noising). The best of these four solutions (It is always solution 11 and it is denoted by B.) takes a value 1 and the value of solution S on the graphic is equal to $\frac{obj(S)}{obj(B)}$.

#### 3.5.2.5 Parameters

The algorithm relies on three main parameters: $\delta$, $r$, $BPR$. The first parameter is the value of increasing the number of processes in the next iteration of algorithm (line 6 in Algorithm 3.6). The value of $\delta$ is set to either $\frac{|P|}{7}$ or $\frac{|P|}{5}$ depending on the size of the instance to solve. Parameter $NR$ is the number of resources used in the noising method (line 8 in Algorithm 3.6). The value of $NR$ is given by $max(\frac{|R|}{2}, 3)$ for all instances. $BPR$ is the number of iterations in exploring the BPR neighborhood. More precisely, it is the number of processes (process p in Algorithm 3.5) that are selected for BPR neighborhood move, i.e. the number of calls of Algorithm 3.5 in each local search iteration (lines 7, 10, 12 in Algorithm 3.6). The BPR neighborhood can be computationally intensive and time consuming and the value of $BPR$ should be controlled. We use $BPR = 100$ for large and $BPR = 300$ for smaller instances. All these parameters are set after much experimental work. The values of all these parameters showed to be more or less irrelevant for B and X instances. On the other hand, the method with the parameters set to these values achieves the best possible results over all ROADEF challenge instances respecting the computational time limit of 5 minutes.

#### 3.5.2.6 Efficiency

The imposed time limit of only five minutes on the total computational time means that efficiency of the method is of paramount importance. Simple data structures for current remaining and safety capacities, current spread for each service, current costs and others, all associated with the single machine and process, render the calculation of the estimations and their updates related to one or several reassignments very efficient. Profiling and code optimization ensured additional 10 fold speed up of the method. The total number of evaluations done in 5 minutes is up to $40 \times 10^6$ for set A instances and from $5 \times 10^5$ to $10 \times 10^6$ for set B and X instances. The algorithm can produce about 10 solutions for set A instances and 2-5 solutions for sets B and X instances in a given 5 minute time frame. In order to obtain better results, both cores of the processor are used with two different methods (possibly with different parameters) running in parallel.

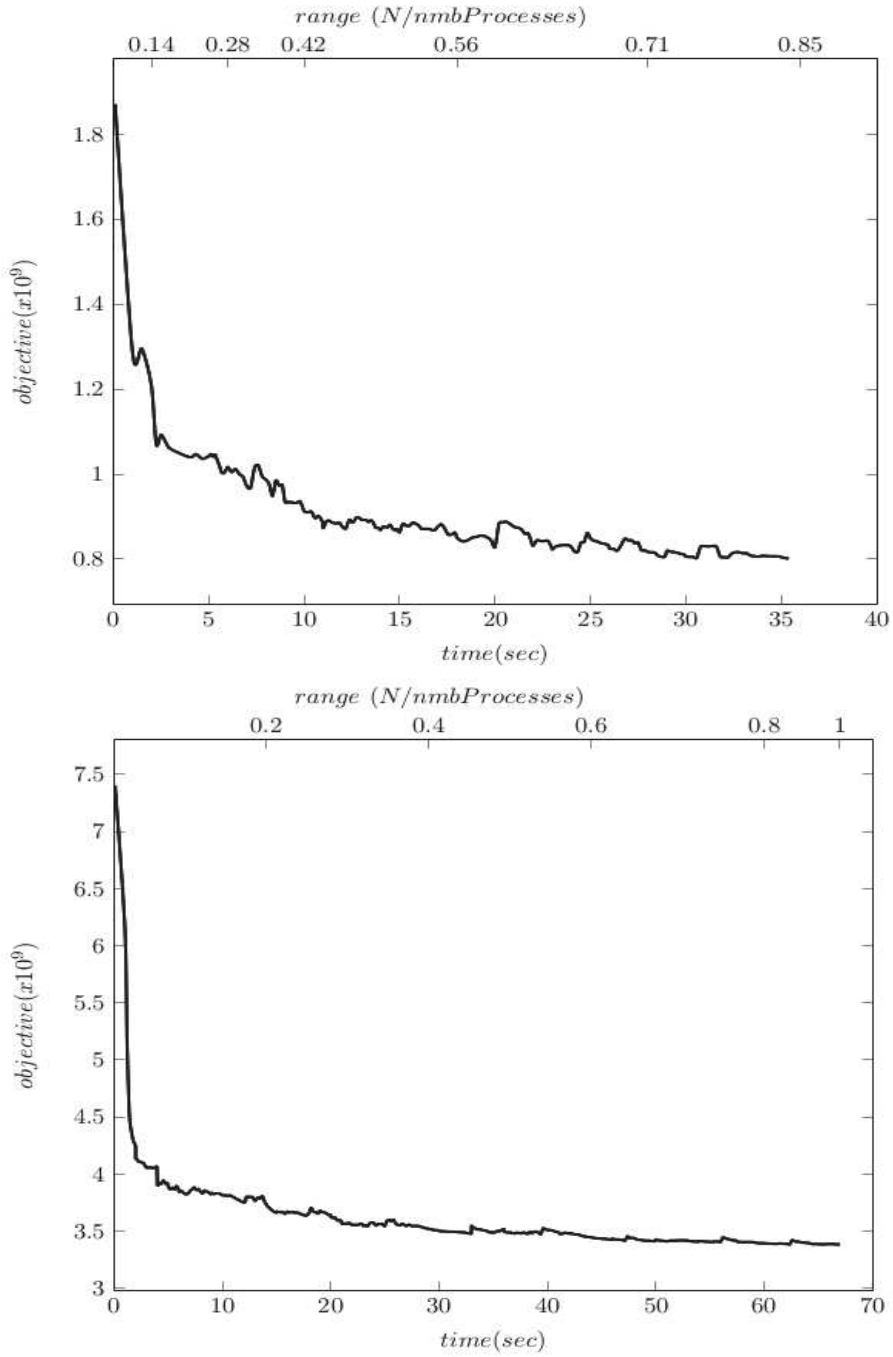


Figure 3.3: The picture ilustrates objective function change during the search for instances $a2\_2$ and $B1$. Bottom x-axis represents CPU time and Y-axis represents objective value. Upper x-asis represents the search range - percentage of processes. To have a real picture about objective change we do not use restarts (few different seeds).

**Suitable data structures** Efficient evaluation of neighborhoods and solution update after performing a move is crucial for the speed of the presented algorithm. Namely, for each of the local search moves, one has to verify (1) the feasibility with respect to given hard constraints and (2) change in the objective function after performing the move. Therefore, aside from the simple data structures that represent the problem data, several redundant data structures have been used in order to reduce the overall computational time, as listed below:

- $MachineResource$ is a structure that stores, for each $(machine, resource)$ pair, a current resource usage on a given machine, with two most important informations being the current remaining capacity on a given resource, $curr\_rem\_capacity$, and current load cost, $curr\_load\_cost$.
- $MachineService$ is $|\mathcal{S}| \times |\mathcal{M}|$ integer matrix, with $MachineService(m, s)$ being the number of processes form service $s$ currently executing on machine $m$. Obviously, all the elements of a matrix are 0 or 1 because of conflict constraint.
- $ServiceLocation$ $|\mathcal{S}| \times |\mathcal{L}|$ integer matrix, with $ServiceLocation(s, l)$ being the number of processes form service $s$ currently executing in location $l$. This matrix is useful when checking spread constraints.
- $ServiceNeighborhood$ - the same as $ServiceLocation$, with neighborhoods instead of locations (useful in dependency constraint checking).
- $ServiceSpread$ is an array representing current number of locations for each service (current service spread).
- $numberOfMovedProcessesFromService$ - a list representing for each service $s$, a number of processes from a service moved from initial machine.
- $numberOfServicesWithMaxNumberOfMovedProcesses$ - the number of services with maximum number of moved processes.

These data structures showed to be useful in evaluating the moves more quickly. Suppose we are moving process $p \in \mathcal{P}$, belonging to service $s \in \mathcal{S}$, to machine

$m \in \mathcal{M}$ belonging to location $l \in \mathcal{L}$ and neighborhood $n \in \mathcal{N}$. Using structure MachineResource, we can verify in $\mathcal{O}(|\mathcal{R}|)$ if the capacity constraints are satisfied. Conflict constraints are satisfied if and only if $MachineService(s, m) = 0$. Spread constraints can be verified in constant time by making sure that the move will not lower the spread of service $s$ below its minimum requirements. This is the case if (1) $ServiceSpread(s) > spreadmin_s$ or (2) $ServiceSpread(s) == spreadmin_s$ and either $p$ is not the only process of service $s$ in its current location or $p$ is the first process of service $s$ in location $l$. Regarding dependency constraints, process $p$ cannot be moved if it is the only process of service $p$ in its current neighborhood and another process in that neighborhood depends on it. Additionally, one has to assure that $n$ contains processes that satisfy all the dependencies of service $s$. For the set of dependencies $D$, all these requirements can be verified in $\mathcal{O}(|D|)$. In summary, the feasibility of a shift move can be verified in $\mathcal{O}(|\mathcal{R}| + |D|)$. The difference between the cost of the new solution and the cost of the current solution can also be computed in $\mathcal{O}(|\mathcal{R}|+|D|)$. When executing the move all data structures must be updated to reflect the new assignment. This can be done in $O(|R|)$. All the other moves i.e. *swap* and $BPR$ can be seen as a sequence of shift moves and the same rules are applied.

### 3.5.3 Final Algorithm

Final algorithm pseudo-code is given in Algorithm 3.6. Since local search procedure ($LS()$) is invoked many times (for each considered $N$ and for $NR$ resources), $BPR$,

*shift* and *swap* neighborhoods are explored only once, as shown in Algorithm 3.7.

**Algorithm 3.6:** NLS

```
1  Sort processes by sum of requirements ;
2  N ← 0;
3  S_best ← S_initial ;
4  S_current ← S_initial ;
5  while N < |P| do
6      N ← N + δ;
7      LS(S_current, N);
8      for i = 1 to NR do
9          Change objective - increase weight of resource i;
10         LS(S_current, N);
11         Set resources weights to original;
12         LS(S_current, N);
13         if cost(S_current) < cost(S_best) then
14             S_best ← S_current
15         S_current ← S_best
```

**Algorithm 3.7:** $LS()$

```
1  input: current solution, number of processes to consider N;
2  BPR();
3  shift();
4  swap();
```

## 3.6 Computational Results

In this section, we present the computational results for a provided set of instances. The benchmark dataset is composed of 3 sets (A, B and X) of 10 instances that were used in the ROADEF/EURO Challenge 2012. The size of these instances ranges from 4 machines and 100 processes to 5,000 machines and 50,000 processes. The instances of set A have a smaller size than the others, while sets B and X are larger and very similar in nature. Despite their smaller size, instances A showed to be

much harder to solve to the optimality (e.g. $a2\_2$ and $a2\_3$ instances). For almost all instances of sets B and X the gap between the solution value and a lower bound is smaller than 0.01%. All the results are obtained on a core2duo E8500 3.16GHz with 4GB RAM on Debian 64 with maximum execution time fixed to 5 minutes for each instance. The same machine is used for ROADEF/EURO Challenge evaluation. The whole method is implemented in C++ and the code itself is available as an open source project. The program was run 100 times for each instance. The average and best objective values are reported. The score function is the one used for challenge evaluation and involves measuring and comparing the cost gains relative to the initial solution. In Tables 3.10 and 3.11, we present our computational results for a provided set of instances. Longer running times may lead to additional solution improvements, but most of the possible improvement is already achieved after five minutes. Therefore, we do not give any solution reports with running time greater than 5 minutes. Using greater time limit is equivalent to multiply runs with different seeds (for example, best reported results in Tables 3.10 and 3.11 can be seen as average solutions in 500 minutes running time).

**Score function** Some instances considered in this work may have arbitrarily low optimal costs. Thus, assessing the solution quality as a gap from the best solution cost would give too much importance to problems for which the expected final solution value is low. We thus rely on another alternative, which involves measuring and comparing the cost gains relative to the initial solution which is quite natural approach and is used for challenge evaluation. The score of solution $S$ respect to solution $B$ is equal to

$$\frac{cost(S) - cost(B)}{cost(originalSolution)}$$

. Improvement can be expressed as a percentage: 0% means no improvement over the original solution and 100% means that the cost of the original solution has been reduced to 0.

| Results A - 100 runs | | | | | | |
|---|---|---|---|---|---|---|
| Inst | Average | Best | Best Q | LB | LP LB | |
| a1_1 | 44 306 501 | 44 306 501 | 44 306 501 | 44 306 390 | 44 306 501 | * |
| a1_2 | 778 142 261 | 777 536 907 | 777 532 896 | 777 530 730 | 777 531 000 | * |
| a1_3 | 583 006 320 | 583 005 818 | 583 005 717 | 583 005 700 | 583 005 715 | * |
| a1_4 | 259 815 285 | 245 421 266 | 252 728 589 | 242 387 530 | 242 406 000 | |
| a1_5 | 727 578 311 | 727 578 309 | 727 578 309 | 727 578 290 | 727 578 000 | * |
| a2_1 | 333 | 199 | 198 | 0 | 126 | * |
| a2_2 | 740 140 535 | 707 237 541 | 816 523983 | 13 590 090 | 537 253 000 | |
| a2_3 | 1 210 207 120 | 1 182 260 491 | 1 306 868 761 | 521 441 700 | 1 031 400 000 | |
| a2_4 | 1 680 629 156 | 1 680 542 520 | 1 681 353 943 | 1 680 222 380 | 1 680 230 000 | |
| a2_5 | 317 804 454 | 309 714 522 | 336 170 182 | 307 035 180 | 307 403 000 | |
| Score | -9.36 | -15.86 | | | | |

Table 3.10: The table shows the results for set A instances. The average and best objective values are reported by running the program for 100 different seeds with 5 minutes running time. The fourth column (Best Q) represents the best solutions from the qualifying phase of competition. The fifth column represents the score of our average (best) solutions with respect to the best solutions from qualification phase and Total Score is the sum of these values. The last two columns represent the solution lower bounds, the first one is simple lower bound described in Section 3.4 and the second one is linear programming based lower bound taken from Masson et al. [2013]. The instance is marked by (*) if the load and balance costs are optimal.

## 3.7 Conclusion

Local search algorithm has been proposed for the Machine Reassignment problem proposed by Google. Local search starts with initial solution given as an input data and improves it by exploring three local search neighborhoods; two commonly used neighborhoods, shift and swap, and one more complex called BPR (Big Process Rearrangement) neighborhood. Several strategies have been developed in order to obtain better solutions including intensification and diversification of the search, defining a good order of processes when exploring the search space, and restart procedures. Noising strategy consisting of slight objective function modification showed to be useful in improving the final results. To deal with the size of the

| Results B, X - 100 runs | | | | |
|---|---|---|---|---|
| Inst | Average | Best | Best Challenge | LB | |
| *B1* | 3 343 410 128 | 3 297 378 837 | 3 339 186 879 | 3 290 754 940 | |
| *B2* | 1 015 561 513 | 1 015 515 249 | 1 015 553 800 | 1 015 153 860 | * |
| *B3* | 157 737 166 | 156 978 411 | 156 835 787 | 156 631 070 | * |
| *B4* | 4 677 981 438 | 4 677 961 007 | 4 677 823,040 | 4 677 767 120 | * |
| *B5* | 923 905 512 | 923 610 156 | 923 092 380 | 922 858 550 | * |
| *B6* | 9 525 934 654 | 9 525 900 218 | 9 525 857 752 | 9 525 841 820 | * |
| *B7* | 14 835 328 102 | 14 835 031 813 | 14 835 149 752 | 14 833 297 940 | * |
| *B8* | 1 214 453 127 | 1 214 416 705 | 1 214 458 817 | 1 214 153 440 | * |
| *B9* | 15 885 693 227 | 15 885 548 612 | 15 885 486 698 | 15 885 064 440 | * |
| *B10* | 18 048 711 483 | 18 048 499 616 | 18 048 515 118 | 18 048 006 980 | * |
| *X1* | 3 065 081 130 | 3 030 246 091 | 3 100 852 728 | 3 023 565 050 | |
| *X2* | 1 003 356 104 | 1 002 698 043 | 1 002 502 119 | 1 001 403 470 | |
| *X3* | 341 508 | 259 656 | 211 656 | 0 | * |
| *X4* | 4 721 856 521 | 4 721 820 325 | 4 721 629 497 | 4 721 558 880 | * |
| *X5* | 160 418 | 144 768 | 93 823 | 0 | * |
| *X6* | 9 546 972 261 | 9 546 967 016 | 9 546 941 232 | 9 546 930 520 | * |
| *X7* | 14 253 212 517 | 14 253 133 805 | 14 253 273 178 | 14 251 967 330 | * |
| *X8* | 147 269 | 138 083 | 42 674 | 0 | * |
| *X9* | 16 125 760 293 | 16 125 746 709 | 16 125 612 590 | 16 125 494 300 | * |
| *X10* | 17 815 072 367 | 17 815 045 320 | 17 816 514 161 | 17 814 534 020 | * |
| Score | -0.38 | -1.48 | | | |

Table 3.11: The table shows the results and lower bounds for set B and X instances. We do not report the score for each instance since all the results are very close to the lower bounds. The total score is equal to -0.38 (-1.48) for average (best) results. The instance is marked by (*) if the load and balance costs are optimal.

problem instances, search exploration is limited for each of the neighborhoods. The proposed solution is still sensitive to the choice of the parameters and the appropriate choice of processes and machines participating in the moves. The computational tests show that these choices in the initial phase of the method greatly influence the quality of the final solution. Nevertheless, some of the obtained results are quasi optimal, while the others are competitive with the world's best known results. The challenge remains to construct essentially different types of local search moves. We believe it would be very useful to design an efficient algorithm to calculate the optimal assignment of processes on two given machines, taking into consideration only the processes already assigned to those machines. While the whole problem is defined as an improvement problem for a given solu-

tion, the construction of an initial solution from scratch would bring a new insight to the data and the solution method which would improve the presented local search itself.

# Chapter 4

# SNCF Rolling Stock Problem

This chapter presents the method developed for solving the problem of Rolling Stock unit management on railway sites, defined by French Railways (SNCF). The problem is very complex and includes several difficult sub-problems. Contrary to the pure local search methods described in the previous chapters, approach proposed herein combines greedy heuristics, Mixed Integer Programming (MIP) and local search. Greedy heuristic (rather complex) and Integer Programming have been used in order to obtain initial feasible solutions to the problem, which are then the subject of an improvement procedures based on local search.

## 4.1 Introduction

The problem of rolling stock unit management at railway sites, as defined by French Railways (SNCF) has been proposed at the ROADEF/EURO Challenge 2014 competition. The problem involves managing trains between their arrivals and departures at terminal stations. This problem is currently being jointly addressed by several SNCF departments, thus decomposing it into a collection of sub-problems to be solved sequentially. Consequently, the integrated problem formulation exposed here in fact reflects a prospective approach. Between terminal station arrivals and departures, the trains never do vanish. This aspect unfortunately is often neglected in railway optimization methods. In contrast, in the past, rail networks possessed sufficient capacity to accommodate all trains without too

many constraints: such is no longer the case. Traffic has indeed increased considerably in recent years, and a number of stations are now experiencing real congestion issues. Current traffic trends will make this phenomenon even more challenging over the next few years. The proposed model focuses on the multiple dimensions of this problem, by taking into account many different aspects. The model scope remains within geographically limited boundaries, typically just a few km in urban environments: the train station and surrounding railway infrastructure resources are considered by this model. The solutions to such problems involve temporary parking and shunting on infrastructure, which typically consists of station platforms, maintenance facilities, rail yards located close to train stations and the set of tracks linking them (these infrastructure resources constitute what is referred to as the "system").
This chapter will be organized as follows. A description of the problem is provided in Section 4.2. The description employed herein has been borrowed from the official competition subject (Ramond and Marcos [2014]). Section 4.3 will address all related work. Our two-phase approach will be described in Section 4.4: solving the problem of matching (assigning) trains to departures will be considered first, followed by the problem of scheduling trains inside the station. An iterative improvement procedure, based on a local search, will be presented at the end of this section. The computational results obtained from the available set of instances, provided by SNCF, will be provided in Section 4.5. Our algorithm developed for solving the preliminary version of the problem used during the qualification phase of the competition, along with computational results, is presented in Section 4.6. The authors' final remarks and conclusion will be given in the last section.

## 4.2 Problem Statement

### 4.2.1 Planning horizon

The planning horizon considered in this problem is variable. It is an integral number ($nbDays$) of days from morning of day 1 ("d1" at 00:00:00, denoted $h0$) to midnight of day $nbDays$ ("dnbDays" at 23:59:59). No absolute date is considered, we assume that the days to be planned can be at any date. All days last 24 hours,

and no time change occurs in the horizon. The set of all time instants within the planning horizon is denoted by $\mathcal{H}$. In the following, we use the word "time" to represent a time instant (date/time) during the horizon. A time within this horizon is written "$d_i\ hh : mm : ss$", where $i \in [1, nbDays]$ stands for the day index, $hh \in [0, 23]$ for the hours, $mm \in [0, 59]$ for the minutes and $ss \in [0, 59]$ for the seconds. This representation implies that the time horizon is discretized, the smallest duration taken into account being one second. Durations have a format similar to time instants ($hh : mm : ss$), but the number of hours $hh$ may be greater than 23. Note that using the second as the shortest time unit enables to represent time with a high precision, considering that most durations used in the following are typically a few minutes or hours. All instances may not fully exploit this precision; some may only handle durations and time instants as multiples of 10 seconds or one minute, for example. Depending on the resolution approach, this might be used to reduce the problem complexity.

### 4.2.2 Arrivals

Arrivals are the end of journeys for passengers. In our model, arrivals generate entrances of trains in the system. The times of arrivals are provided as an input and are considered to be non-modifiable. These times correspond to the moments trains arrive on platforms, but their entrance in the system usually occurs a few minutes before. Indeed, trains have to perform what is called "arrival sequences". These sequences are fixed and represent the routing of trains on the tracks during the last few km before platforms. Arrival sequences are non-modifiable but they require some resources of the station, such as a set of tracks during pre-defined time periods, and in that sense they impact the efficiency and the usage of the whole system. Formally, the set of arrivals during the horizon is denoted by $\mathcal{A}$ and an arrival $a \in \mathcal{A}$ is defined by the following characteristics:

- the associated train, $arrTrain_a$ (see the set $\mathcal{T}$ of trains introduced in Section 4.2.4),
- the arrival time $arrTime_a$ ,

- the arrival sequence $arrSeq_a$ , which is the sequence of track groups (see Section 4.2.7.5) used immediately before arriving on platform,
- the set of preferred platforms, $prefPlat_a$ , determining which platforms are preferred to be assigned to $a$ (these preferences may be unsatisfied in solutions, but this is penalized by a cost in the objective function),
- the ideal dwell time, $idealDwell_a$ , and the maximum dwell time $maxDwell_a$, which respectively represent the ideal and the maximum time $arrTrain_a$ should stay on the platform after $arrTime_a$ before moving to some other resource,
- the remaining distance before maintenance ($DBM$) $remDBM_a$ and time before maintenance ($TBM$), $remTBM_a$ of $arrTrain_a$, determining whether or not $arrTrain_a$ must perform maintenance operations before being assigned to a departure. In the following, if train $t \in \mathcal{T}$ is associated with $a$ ($arrTrain_a = t$), we define $remDBM_t = remDBM_a$ and $remTBM_t = remTBM_a$.

  Potentially, $a$ can be part of a joint-arrival denoted by $jointArr_a$ . A joint-arrival defines a combination of trains, physically assembled and arriving together at the station. More details on joint-arrivals and joint-departures are provided in Section 4.2.5.

  One decision to make concerning an arrival $a$ is the platform to assign, i.e. the platform on which $arrTrain_a$ arrives at $arrTime_a$. It is feasible (but penalized) not to cover some arrival $a \in \mathcal{A}$. In this case, the associated train $arrTrain_a$ merely does not come into the system and, hence, does not use any resource during the planning horizon. Obviously, in such cases no platform has to be assigned to $a$. Another direct consequence is that $arrTrain_a$ cannot be assigned to any departure.

### 4.2.3 Departures

Departures are known by passengers as the beginning of a train journey. From the problem's perspective, departures are the way trains leave the system. As for arrivals, a platform must be assigned to each departure; their times as well

as their departure sequences (routing between platforms and the limits of the system) are known, fixed and given in advance. One has to decide which train is assigned to each departure. Assigning no train to a departure is highly undesirable; uncovered departures constitute a significant part of the objective function (see section 4.2.10). Note that at most one train can be assigned to a departure. The train assigned to departure $d$, if any, is denoted by $depTrain_d$. The following attributes define a departure $d \in \mathcal{D}$, where $\mathcal{D}$ represents the set of all departures:

- its departure time $depTime_d$,
- its departure sequence, $depSeq_d$ , which is the sequence of track groups used immediately after departing from platform,
- the set of preferred platforms, $prefPlat_d$, determining which platforms are preferred to be assigned to $d$,
- the set of compatible train categories, $compCatDep_d$, which is a subset of $\mathcal{C}$ (see section 4.2.4.2) determining which trains can be assigned to $d$ (only trains whose category is in $compCatDep_d$ can be assigned),
- the ideal dwell time, $idealDwell_d$, and the maximum dwell time, $maxDwell_d$, which respectively represent the ideal and the maximum time the assigned train should stay on the platform before $depTime_d$,
- the distance, $reqDBM_d$ and time $reqTBM_d$ of the journey following the departure. These two values are compared, for a train $t \in \mathcal{T}$ , with the remaining $DBM$ and $TBM$ of $t$, to determine whether or not maintenance operations have to be performed on $t$ before $depTime_d$.

Potentially, d can be part of a joint-departure represented by $jointDep_d$.

This description assumes that the assignments of trains to departures have no impact on arrivals and, more precisely, on the remaining $TBM$ and $DBM$ of trains associated with arrivals. In practice, this is not completely true because the trains associated with arrivals are usually trains which were earlier assigned to departures and which spent some time out of the system before coming back. To

take this into account, some arrivals are linked with departures occuring earlier in the horizon. For such an arrival $a \in \mathcal{A}$, we introduce an additional parameter denoted by $linkedDep_a \in \mathcal{D}$. $linkedDep_a$ is the departure whose assigned rolling stock unit comes back in the system associated with $a$. Then, if a train $t$ is assigned to $d$, and if $a$ is covered, the default train category of $arrTrain_a$ provided in the input data is replaced by the train category of $t$. In a similar way, the default remaining $DBM$ and $TBM$ of a are replaced by values depending on $t$(see Section 4.2.6).

### 4.2.4 Trains

We consider a set of trains denoted by $\in \mathcal{T}$ . In our model, we define a train as a visit in the system of a rolling stock unit. The set of trains is composed of:

- trains already present in the system, located on one of its resources at the beginning of the horizon (this set is represented by $\mathcal{T}_I \subset \mathcal{T}$),
- trains associated with arrivals (these trains belong to set $\mathcal{T}_A \subset \mathcal{T}$).

Rolling stock units are unoriented, composed of railcars which may not be decomposed nor recombined with those of other units. In fact, railcars are not considered in this model: trains are the smallest rolling stock elements.
Today, almost all modern rolling stocks are reversible and non-decomposable units: high-speed trains, recent regional trains, etc. But older trains still have a composition that varies depending on the destinations and weekdays; this implies different numbers and types of railcars, 1st vs 2nd class repartition... To keep the problem description simple, we don't handle this aspect here. The successive stays of a given rolling stock unit in the system during the horizon are considered as different trains in this problem. For instance, a unit may be assigned to a departure $d \in \mathcal{D}$, leave the system and return back a few days later in it, associated with an arrival $a \in \mathcal{A}$: in this case, we consider two distinct trains. As described earlier, these trains might correspond to the same rolling stock unit, if $linkedDep_a = d$, but the trains $arrTrain_a$ and $depTrain_d$ are different because they correspond to distinct visits in the system.

As a consequence, trains are considered only during a portion of the planning horizon: between their associated arrival (or the beginning of the horizon, if they are initially in the system) and the departure they are assigned to (or, if they are not assigned to any departure during the horizon, the end of the horizon).
A train $t \in \mathcal{T}$ belongs to a train category, $cat_t \in \mathcal{C}$, defining some common technical characteristics, such as length, that $t$ shares with other trains of the same category, $\mathcal{C}$ being the set of all train categories.

#### 4.2.4.1 Trains initially in the system

As mentioned earlier, some trains can be present in the system at the beginning of the planning horizon. In addition to its train categories, a train $t \in \mathcal{T}_I$ initially in the system is characterized by the following attributes:

- $res_t$: the resource used by $t$ at $d100:00:00$,
- $remDBM_t$: the remaining distance before maintenance of $t$,
- $remTBM_t$: the remaining time before maintenance of $t$.

Like arrivals which can remain uncovered, one may choose not to use some trains initially in the system, which is feasible but penalized. The unused initial trains, if any, do not use any resource of the system during the planning horizon and cannot be used to be assigned to departures. It is assumed that trains initially in the system are not travelling on track groups at $h0$.

#### 4.2.4.2 Train categories

As introduced in Section 4.2.4, trains share common characteristics defined by their belonging to some categories. In a sense, all trains belonging to the same category are identical, with only their initial maintenance conditions (remaining $TBM$ and $DBM$) being different from one train to another. In practice, trains are usually produced in series of a few dozen or hundred identical units which are eventually delivered by the manufacturers to the railways companies: these A train category $c \in \mathcal{C}$, where $\mathcal{C}$ is the set of all train categories, is defined by:

- a length: $length_c$,
- a compatibility group $catGroup_c$ expressing the physical and technical compatibility of trains with each other when two or more trains are assembled: two trains are compatible if and only if their respective compatibility groups are identical,
- a maximal distance before maintenance, $maxDBM_c$, expressed in $km$ and defining the $DBM$ of trains belonging to category $c$ once they finish a maintenance operation of type "D"(restoring the $DBM$, see 4.2.6). This quantity represents the maximal number of $km$ a train of category $c$ can run between two maintenance operations of type "D".
- similarly, $maxTBM_c$ designates the maximal time before maintenance of trains of category $c$. This is the maximal time trains of category $c$ can run between two maintenance operations of type "T".
- the time $maintTimeT_c$ required to perform a maintenance operation of type "T", and
- the time $maintTimeD_c$ required to perform a maintenance operation of type "D".

For the sake of simplicity, in the following we use the following convention (the characteristics of a train refer to those of its category). For any $c \in \mathcal{C}$ and any $t \in \mathcal{T}$ such that $cat_t = c$, we set:

$$
\begin{aligned}
length_t &= length_c \\
catGroup_t &= catGroup_c \\
maxDBM_t &= maxDBM_c \\
maxTBM_t &= maxTBM_c \\
maintTimeT_t &= maintTimeT_c \\
maintTimeD_t &= maintTimeD_c
\end{aligned}
$$

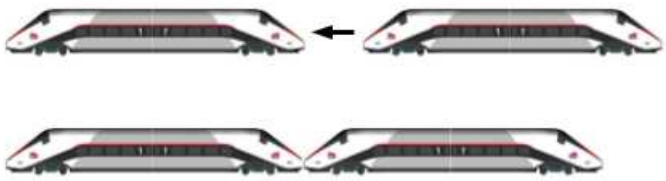

Figure 4.1: Junction of two trains

#### 4.2.4.3 Preferred train reuses

From a practical point of view, the assignment of trains to departures is not always completely free. Indeed, in order to ensure the feasibility of the schedules, some decisions are planned in an earlier stage; train reuses are part of these decisions. They consist in pre-determining which arriving train is supposed to be assigned to a given departure. This might be sub-optimal in some particular situations, especially when changes occur in the tactical planning and some assumptions do not hold any-more. However, this kind of information is shared among a high number of workers and these decisions are difficult to change because they induce disorganization in the operational management.
To represent this, some preferred train reuses are provided as an input in some instances. A reuse $u \in U$, where $U$ denotes the set of all reuses, is defined by an arrival, $arr_u$ , and a departure, $dep_u$ . For such a reuse $u$, when possible, the train assigned to $dep_u$ should be the same, ideally, as the one associated with $arr_u$ . This might not be respected, but in this case a non-satisfied reuse cost applies in the objective function. In particular, this cost applies if any of $arr_u$ or $dep_u$ is uncovered. It is assumed that any departure $d \in \mathcal{D}$ appears in at most one reuse, i.e. there is at most one reuse $u$ such that $dep_u = d$. Conversely, for any $a \in \mathcal{A}$ there is at most one reuse $u$ such that $arr_u = a$.

### 4.2.5 Joint-arrivals and joint-departures

When the number of expected passengers is high, some commercial trips are covered by more than one train:$n \geq 2$ trains are physically assembled to run together. We call a joint-arrival a combination of $n$ simultaneous arrivals and a

joint-departure the equivalent for $n$ simultaneous departures, corresponding to $n$ assembled trains arriving to or departing from the same platform. A joint-arrival concerns $n$ arrivals, and expresses a synchronization of arrivals. A joint-departure consists of $n$ departures, all having the same time; the associated trains must be assembled, on the same platform, in a certain order. Indeed, the roles played by the different trains differ according to the departure they are assigned to, and their order is important when they are disassembled, even when this happens outside of the considered system. Similarly, all the trains of a joint-arrival share the same time and the order of the arriving trains on the platform is an attribute of the joint-arrival. Formally, a joint-departure $j \in \mathcal{J}_{dep}$ , where $\mathcal{J}_{dep}$ is the set of all joint-departures, is defined by an ordered list of departures $jdList_j$. In a symmetrical way, $J_{arr}$ represents the set of all joint-arrivals and any $j \in \mathcal{J}_{arr}$ is characterized by an ordered list of arrivals $jaList_j$. The order of arrivals/departures is important because it enables to distinguish which train (associated with which departure/arrival) runs first, at the head of the "convoy". The convention adopted throughout this document is as follows: the first departure/arrival in the lists $jdList_j$ and $jaList_j$ is associated with the train located most on side $A$ of the assigned platform (as explained later in Section 4.2.7, all resources have a side $A$ and a side $B$), and the next departures/arrivals are associated with the next positions. The junction of one train with another train, or with already assembled trains, is an operation which has a cost denoted $junCost$ and requires a duration denoted by $junTime$. This operation is presented in Figure 4.1. Starting from trains not assembled, the junction of $n$ trains cannot be performed in a single junction operation: it requires $n-1$ junction operations and, consequently, costs $(n-1) \times junCost$ and requires a duration of at least $(n-1) \times junTime$. Junction operations can be performed only on platforms, single tracks, maintenance facilities or yards (see types of resources in Section 4.2.7). Once assembled, the trains are considered as if they were a single train when they move on track groups: only one move is considered when assembled trains run on a track group. However, they are still considered as multiple trains on the other types of resources.

Symmetrically, a disjunction of trains has a cost $disjCost$ and requires some time represented by $disjTime$. Like junctions, disjunctions can only be performed on platforms, single tracks, maintenance facilities or yards. A disjunction operation

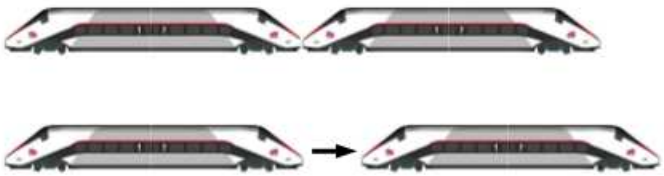

Figure 4.2: Disjunction of two trains

is shown in Figure 4.2. The schedules associated with assembled trains should be the same between their junction and disjunction: beginning times on each resource as well as resources themselves must be identical as long as trains are assembled.

### 4.2.6 Maintenance

Trains must be maintained on a regular basis to be able to run in proper security and comfort conditions. Their ability to be assigned to departures is determined by comparing their $DBM$ and $TBM$ with the requirements of the departure (distance and time required for the journey following the departure). The $TBM$ is a value representing the time a train can run before a maintenance of type "T" needs to be performed. It is mostly associated with comfort considerations. The DBM is more related to security constraints. It represents the maximum distance a train can still run before the next maintenance of type "D". For a given train performing no maintenance operation in the system, the $DBM$ and $TBM$ remain unchanged between an arrival and a departure: local moves between resources are neglected as their speed is slow and the associated distances are short with respect to those induced by departures; the time spent in the system by the train is not considered as affecting the $TBM$. Two types of maintenance operations may be performed: maintenance of type "D" enables to restore the $DBM$ of a train t to its maximum value, $maxDBM_t$, whereas maintenance of type "T" is its equivalent for time (restores $TBM$ to its maximum value, $maxTBM_t$). A train may perform at most one operation of each type. Indeed, subsequent operations would have no effect on $TBM/DBM$ because $TBM/DBM$ would already be restored at their maximum value. When an arrival $a$ has a linked departure $d$ (i.e. $linkedDep_a = d$), two cases must be distinguished. If $d$ is not covered (i.e. if no train is assigned to $d$), then

the remaining $TBM$ and the remaining $DBM$ of a are those provided in the input data for $a$. If $d$ is covered, these characteristics are replaced by those induced by $depTrain_d$: the remaining $TBM$ and $DBM$ of a are deduced from those of the train assigned to $d$. If $d = linkedDep_a$, $t = depTrain_d$ and no maintenance operation is performed on $t$, we have

$$remDBM_a = remDBM_t - reqDBM_d$$
$$remTBM_a = remTBM_t - reqTBM_d$$

If a maintenance operation of type "D" is performed on $t$, then

$$remDBM_a = maxDBM_t - reqDBM_dm$$

and if a maintenance operation of type "T" is performed on $t$,

$$remTBM_a = maxTBM_t - reqTBM_d.$$

For instance, if a train $t$ has remaining $DBM$ of 500km and a remaining $TBM$ of 48 hours, it can be assigned to a departure requiring a $DBM$ of 450km and a $TBM$ of 24 hours. However, it may not be assigned to a departure requiring a $DBM$ of 450km and a $TBM$ of 52 hours (this would violate the $TBM$ requirement). It may also not be assigned to a departure requiring a $DBM$ of 550km and a $TBM$ of 24 hours (this would then violate the $DBM$ requirement).
Maintenance operations can only be performed on maintenance facilities (described in section 4.2.7.4) which are dedicated to only one type of maintenance. Consequently, a train may not perform both types of operations on the same maintenance facility. The duration of maintenance operations depends on the category of trains. Maintenance capacities in the system being limited, the number of maintenance operations which can be performed over a day in the system, i.e. over all maintenance facilities of the system, is bounded by a maximal value represented by $maxMaint$.

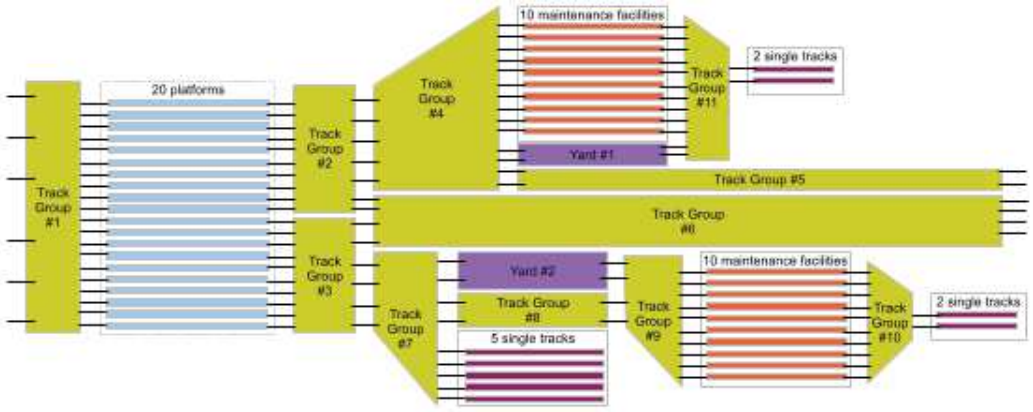


Figure 4.3: Example of resources infrastructure

### 4.2.7 Infrastructure resources

Between arrivals and departures, trains are either moving or parking on tracks that we consider as resources. In practice, trains can be very long and occupy more than only one track at a time, but we neglect this aspect here, we consider trains as if they were points which can instantly move from one resource to another.
Let $\mathcal{R}$ be the set of all resources. Resources can be either single tracks, platforms, maintenance facilities, yards or track groups; $\mathcal{S}$, $\mathcal{P}$, $\mathcal{F}$, $\mathcal{Y}$ and $\mathcal{K}$ represent respectively the set of all single tracks, all platforms, all maintenance facilities, all yards and all track groups of the system. Single tracks, platforms and maintenance facilities represent portions of tracks considered in an individual manner, while track groups and yards are aggregated types of resources which usually contain more than only one track and contain switches to physically link the different tracks together. The next sections provide details for each of these types of resources. An example of resources configuration associated with a sample system are presented on Figure 4.3. Infrastructures corresponding to dataset A (used in qualification stage) and some instances in datasets B and X (used in the final stage of competition) are illustrated in Figures 4.4 and 4.5.

#### 4.2.7.1 Transitions between resources

A resource $r \in \mathcal{R}$ has a set $neighSet_r$ of neighbor resources, defining the possible transitions for trains, in a symmetrical way: if a resource $r'$ is a neighbor of $r$,

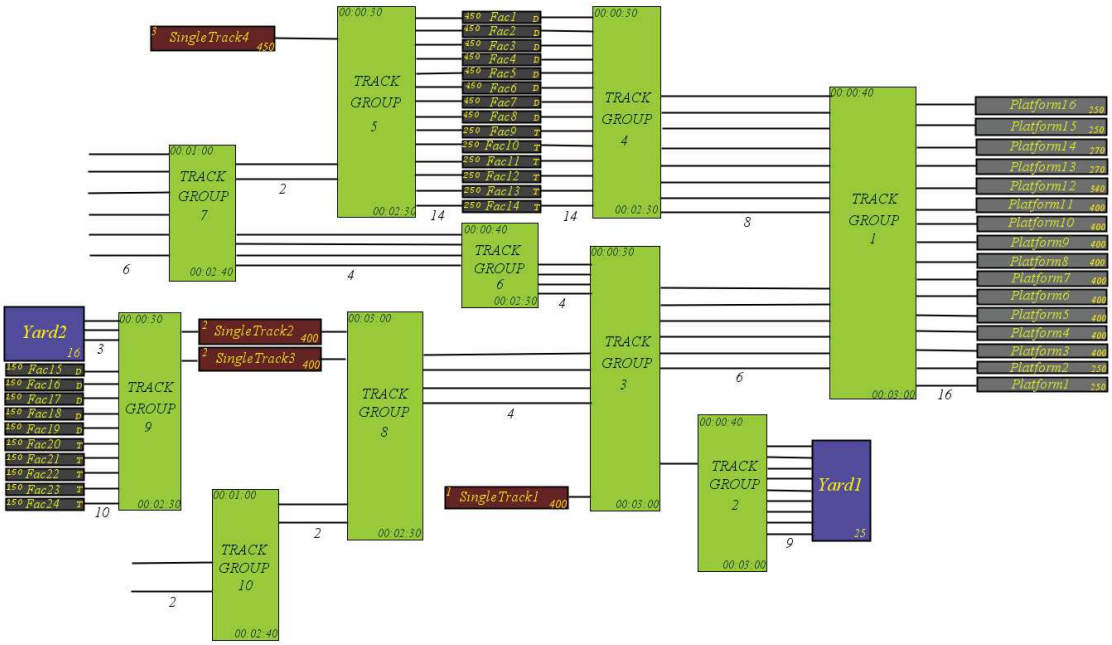


Figure 4.4: Infrastructure for instances A1 – A6

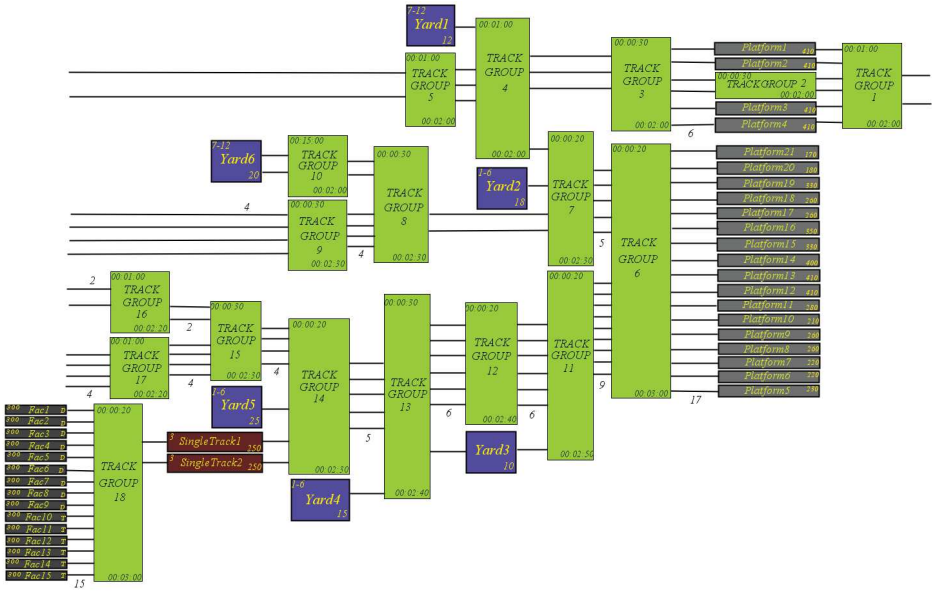


Figure 4.5: Infrastructure for instances A7 – A12

that is if it belongs to $neighSet_r$, $r$ is a neighbor of $r'$ and a train might use $r'$ immediately after $r$ (and vice-versa). On the contrary, if $r'' \notin neighSet_r$ , the direct transition from $r$ to $r''$ is not allowed (and neither from $r''$ to $r$): an intermediate resource must be used between $r$ and $r''$. As we are interested only in transitions between different resources, a resource is not a neighbor of itself. In our model, resources represent railways infrastructure elements which are in general linear and can be accessed from at most two sides, and in some cases from only one side. Given this aspect, the neighbors of a resource can then be divided into two subsets, one being physically associated with each "side" of the linear element. By convention, these two sides are denoted $A$ and $B$. Hence, for any resource $r$, the set of neighbors of $r$ is decomposed into two subsets:

$$neighSet_r = neighSet_r^A \cup neighSet_r^B$$

. It is also assumed that a resource cannot be the neighbor of another resource both on $A$ and $B$ sides:

$$neighSet_r^A \cap neighSet_r^B = \emptyset.$$

The transitions between a resource, $r$, and one of its neighbors, are performed through one of the entry/exit points, which we call *gates*, located on both sides of the resource. These gates correspond to the physical tracks linking the different resources. Let $G_r$ denote the set of gates of resource $r$. A gate $g \in G_r$ is defined by its side $side_g \in \{A, B\}$ and its index $ind_g \in N$ ; $r_g$ represents the resource $g$ belongs to. The neighbor gate of $g$, $neigh_g$ , is unique and represents the gate of a neighbor resource accessible through $g$. The relation between a gate and its neighbor gate is reflexive: the neighbor gate of $neigh_g$ is $g$. The only exception concerns the gates at the boundaries of the system, which do not have a neighbor. Trains must run through these gates without neighbor to enter or exit the system. On each side, the gates are ordered according to their physical positions. The indices of the gates follow this order. For instance, if a track group consists of tracks oriented following an East-West axis, the gates on each side are ordered from North to South. On side $A$, the gate most on the North is called $A1$, the next gate $A2$, and so on.


Figure 4.6: Example of gates in track group

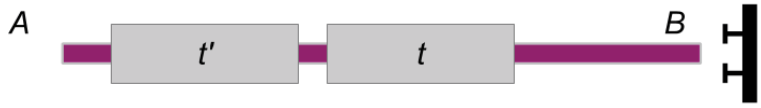


Figure 4.7: Example of resource with only one gate, on side $A$

Figure 4.6 presents an example for a track group. Associated with each gate of a particular resource is exactly one gate associated with one of its neighbors (or no neighbor if the gate represents is at the boundary of the system). Single tracks, platforms and facilities, representing individual tracks, have at most one gate on side $A$, and at most one gate on side $B$ (hence, at most one neighbor resource on each side):

$$\forall r \in \mathcal{S} \cup \mathcal{P} \cup \mathcal{F}, |neighSet_r^A| \leq 1 \wedge |neighSet_r^B| \leq 1.$$

When only one side is accessible, for instance a platform in a terminal station where the track ends, we use the convention that only side $A$ is accessible: $neighSet_r^A \neq \emptyset \wedge neighSet_r^B = \emptyset$.
A resource $r \in \mathcal{S} \cup \mathcal{P} \cup \mathcal{F}$ with only one gate on side $A$ and no gate on side $B$ can be seen as a stack. It must be managed in a Last In First Out (LIFO) way. This means that a train $t$ cannot leave $r$ at $h \in \mathcal{H}$ if another train $t'$ has arrived later on $r$ and has not yet left $r$. A resource $r \in \mathcal{S} \cup \mathcal{P} \cup \mathcal{F}$ with one gate on side $A$ and one gate on side $B$ can be seen as a double-ended queue. A train $t$ cannot leave $r$ through the gate on side $A$ if another train $t'$ has arrived later through the gate of side $A$.
Only track groups and yards may have more than one gate on each side. For these types of resources, two adjacent resources might be linked by more than one gate: the gate used for the transition between two resources has to be specified.

Some resources can be used only by some particular train categories. The set of train categories compatible with resource r is denoted by $compCatRes_r$. For instance, some infrastructure resources might not be electrified, which prevents their use by electrical rolling stock; in general, maintenance facilities are dedicated to some categories only.

For any train $t$ and any resource $r$ used by $t$ (except if $r$ is a track group), the time difference between the associated EnterResource and ExitResource events must be greater than or equal to a constant duration represented by $minResTime$. To take into account that some operations must be performed on a train to change its direction (in particular, drivers must walk to go to the other extremity of the train), a train entering on a single track through a given side must stay a minimum amount of time on it before going back through this same side. This time is represented by $revTime$.

#### 4.2.7.2 Single tracks

Some tracks of the system, which are not located in stations (i.e. which do not allow boarding and unboarding of passengers and, hence, cannot be used for arrivals or departures), are considered individually, not part of yards or track groups. They are called single tracks. $\mathcal{S} \subset \mathcal{R}$ is the set of single tracks. A single track resource $s \in \mathcal{S}$ has a length $length_s$ and a capacity $capa_s$. At any time $h \in \mathcal{H}$, both the total length of trains and the number of trains parked on $s$ must not exceed the length of the track and its capacity. The order of trains on single tracks must be consistent with their moves. As trains cannot fly over each other, they must respect the order of their arrival on the track to leave it.

#### 4.2.7.3 Platforms

In our model, platforms represent tracks within the train station where passengers can board and unboard the trains. They are very similar to single tracks in the sense that the order of trains must be consistent with their respective times of moves. However, they do not have a capacity expressed in maximal number of trains. Moreover only platforms can be assigned to arrivals and departures.

$\mathcal{P} \subset \mathcal{R}$ denotes the set of all platforms. Each platform $pin\mathcal{P}$ has a length$length_p$. If the duration of use of a platform is too short before a departure $d \in \mathcal{D}$, less than $idealDwell_d$, passengers might not be able to board the train in a comfortable way. This is penalized by a cost. Symmetrically, trains staying more than $idealDwell_d$ on platforms before departure are penalized because platforms are considered as critical resources. In any case, trains may not use a platform for more than $maxDwell_d$ before departure. Similar considerations apply to arriving trains, which should stay on platforms a duration close to $idealDwell_a$, and in any case less than $maxDwell_a$.
If a platform is temporarily used for purposes other than an arrival or a departure then the duration of use of the platform must not exceed a constant duration denoted by $maxDwellTime$.

#### 4.2.7.4 Maintenance facilities

Maintenance facility resources are special tracks inside maintenance workshops. They are used to periodically reset the $DBM$ and $TBM$ of trains. The set of maintenance facilities is denoted by $\mathcal{F} \subset \mathcal{R}$. A maintenance facility $f \in \mathcal{F}$ is characterized by a type $type_f \in \{"D","T"\}$ indicating which type of operations can be performed. If $type_f$ equals "D", only operations of type "D" can be performed; otherwise, if $type_f$ equals "T", only operations of type "T" can be performed. As a consequence, either the $DBM$ or the $TBM$ is restored by a maintenance facility resource, in an exclusive manner. It is also characterized by a length $length_f$ which may not be exceeded by the total length of trains using it.

#### 4.2.7.5 Track groups

Track groups are sets of tracks used by trains to move throughout the system. A track group represents a sub-part of the rail network in the considered system. Its real physical configuration in terms of tracks and switches linking them can be very complex; we don't consider this complexity here, we rather see it as a black box with some indications on how to identify conflicts. The set of all track groups is represented by $\mathcal{K} \subset \mathcal{R}$. A track group $k \in \mathcal{K}$ is supposed to be used for train

moves: the duration of use of k by any train (i.e. travel time) is a constant denoted by $trTime_k$. It is the time required by a train to enter the track group k on one side and exit at the opposite side. Indeed, a train entering on one side of a track group must exit on the other side. All gates of one side are reachable from all gates of the opposite side. As a consequence, if a track group has $n$ gates on one side and m gates on the other side, then $n \times m$ different paths are possible in each direction. Besides, $hwTime_k$ represents the headway of the track group: this is a security time which must be respected at any place between two trains. Figure 4.6 shows an example of track group representation with $n$ gates on side $A$ and $m$ gates on side $B$. When several trains use a track group over the same time period, conflicts might occur between them. A conflict is an unwished situation where two running trains might come too close to each other, and one has to stop one of them to respect security distances. Once again, we adopt here a "black-box" approach where the full complexity of the track group is eluded. Real conflicts should be identified through a very detailed description of all infrastructure equipments and signaling systems; here, conflicts model non-robustness, that is trains are likely to stop due to headway constraints. Conflicts are not considered feasible. The associated constraints are exposed in Section 4.2.9.

#### 4.2.7.6 Yards

A yard is also a set of tracks, but it is mainly used for parking. Therefore, contrary to track groups which are dedicated to train moves, the duration of use by trains is not fixed, trains can stay on yards with no time restriction. The set of yards is denoted by $\mathcal{Y} \subset \mathcal{R}$. A yard $y \in \mathcal{Y}$ has a limited physical capacity on the number of trains which can be handled simultaneously, $capa_y$, which is a way to model the number of tracks and the maximal number of trains that can be parked simultaneously.

#### 4.2.7.7 Initial train location

It is assumed that trains initially in the system are not traveling on track groups at $h0$ (in other words, for any $t \in \mathcal{T}_I$, $res_t \notin \mathcal{K}$). Moreover, we assume that

when several trains are initially in the system at the beginning of the horizon on the same resource of type single track, platform or maintenance facility, they are positioned by order of appearance in the data input file, starting from side $A$ (the first train appearing in the file is the most on side $A$, the next ones are on the next positions towards side $B$).

#### 4.2.7.8 Imposed resource consumptions

Some aspects of the system we consider are external to our decision perimeter. They are considered fixed and cannot be changed. For instance, some trains use resources during the horizon whereas they are not part of $\mathcal{T}$: infrastructure maintenance trains, trains not terminating at the train station and continuing their journey, or trains from other companies, on which no decision is to be made. They should not be considered the same way in the sense that no decision should be made for them. However they do use the same resources as those of $\mathcal{T}$, i.e. resources of $\mathcal{R}$. Moreover, some resources might be unavailable due to opening times of resources or infrastructure maintenance works. To represent this, pre-determined consumptions of resources are imposed over the horizon on the different resources. Depending on the type of resource, these imposed consumptions have different characteristics. The set of imposed resource consumptions is represented by $\mathcal{I}$. An imposed consumption $i \in \mathcal{T}$ refers to an associated resource, denoted $res_i$. If $res_i$ is an individual track (single track, platform or maintenance facility), it is considered unavailable between a beginning time, $beg_i$, and an end time, $end_i$. No train is allowed to use $res_i$ during the interval $[beg_i, end_i]$. If $res_i$ is a yard $y \in \mathcal{Y}$, then $i$ is defined by a number of trains, $nb_i$ , using the yard $y$ between $beg_i$ and $end_i$. It is equivalent to a temporary reduction of capacity of the yard by $nb_i$ units. We assume that two distinct imposed consumptions for the same yard do not overlap. Finally, if $res_i$ is a track group $k \in \mathcal{K}$, $i$ represents the move of a train over the track group; it is then defined by an origin gate, $o_i$, a destination gate, $d_i$, and the time the train enters the track group at $o_i$, $h_i$. These imposed train paths over track groups are considered exactly the same way as for trains in $\mathcal{T}$ when detecting conflicts.

### 4.2.8 Solution representation

A solution to the problem is composed of a set of schedules, each denoted by $sched_t$ , one for each train $t \in \mathcal{T}$. The schedule $sched_t$ of train $t$ is a sequence of events during its presence in the system, along with details such as the time of each event, the resources used, etc. With this information for every train, it is possible to derive the status of the system and each of its resources at any time during the horizon. Trains in $\mathcal{T}$ which are associated with uncovered arrivals, or unused initial trains must have any empty schedule (no event at all).

The considered events concerning a train are its arrival and departure, its entrance in and exit from the system or any resource of the system, and the beginning and end of junction, disjunction and maintenance operations. These types of events are respectively denoted by Arrival, Departure, EnterSystem, ExitSystem, EnterResource, ExitResource, BegJunction, EndJunction, BegDisjunction, EndDisjunction, BegMaintenance and EndMaintenance. With any event $e \in E$, where $E$ represents the set of all events in the solution, is associated a train, $t_e$, a time, $h_e$, an event type $y_e$, a resource, $r_e$, a gate on $r_e$, $g_e$, and a complement, $c_e$. Depending on the type of event, some of these characteristics may not be relevant.

A solution is feasible if and only if all constraints are satisfied. While most of them were introduced earlier in the previous sections, the constraints of the problem are defined in a more formal way in the competition subject (Ramond and Marcos [2014]).

In the following, the notation $\mathcal{T}^+$ will be used to designate trains which are actually used in the solution, i.e. trains of $\mathcal{T}$ not associated with uncovered arrivals or unused initial trains. Recall that these latter do not use any resource of the system during the whole horizon, so the following constraints do not hold for them.

### 4.2.9 Conflicts on track groups

Depending on their respective times, conflicts may occur between two moves $m_1$ and $m_2$ on a track group. These conflicts are not considered feasible. Moves $m_1$ and $m_2$ can be associated with consecutive EnterResource and ExitResource events in the schedule of trains of $\mathcal{T}^+$, or imposed consumptions on a track group. Let $\mathcal{M}$ represent the set of all moves on track groups. For any move $m_i$ on track group

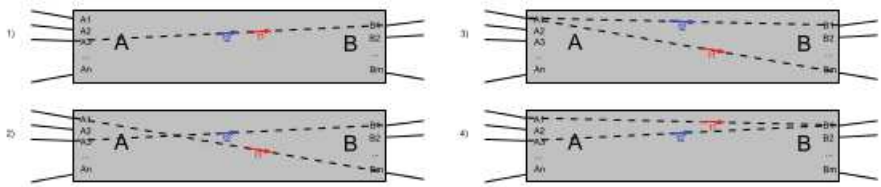


Figure 4.8: Conflicts in the same direction

$k$, let $o_i, d_i$ and $h_i$ respectively denote its origin gate (i.e. the gate through which $m_i$ enters $k$), its destination gate (i.e. through which mi exits $k$) and the time it enters the track group at $o_i$. It is recalled that $trTime_k$ represents the travel time of a train on $k$, $hwTime_k$ the headway time on $k$, meaning the minimum buffer time between two trains to respect security distances, and that assembled trains entering a track group count for only one move. Let $m_1$ and $m_2$ be two moves on the same track group. We consider that a conflict occurs in the following cases:

- Case 1: $m_1$ and $m_2$ are on intersecting paths, in the same direction, with insufficient buffer time (see Figure 4.8 for examples of configurations where conflicts might arise between two trains in the same direction)
  - the indices of destination gates of $m_1$ and $m_2$ are equal, or inverted with respect to those of origin gates:
    $(ind_{o_1} - ind_{o_2}) \times (ind_{d_1} - ind_{d_2}) \leq 0$,
  - $m_1$ and $m_2$ are in the same direction ($side_{o_1} = side_{o_2}$), and
  - the headway time may not be respected ($|h_1 - h_2| < hwTime_k$).
- Case 2: $m_1$ and $m_2$ are on intersecting paths, in opposite directions, with insufficient buffer time
  - the indices of the gates used by $m_1$ and $m_2$ on each side are equal, or inverted with respect to those used on the other side:
    $(ind_{o_1} - ind_{d_2}) \times (ind_{d_1} - ind_{o_2}) \leq 0$,
  - $m_1$ and $m_2$ are in opposite directions ($side_{o_1} \neq side_{o_2}$), and
  - the headway time may not be respected ($|h_1 - h_2| < trTime_k + hwTime_k$).

With this definition, the constraints on conflicts are expressed by:

$$\begin{aligned}
&\forall(m_1, m_2, k) \in \mathcal{M}^2 \times \mathcal{K}\ s.t.\ side_{o_1} = side_{o_2}, \\
&(ind_{o_1} - ind_{o_2}) \times (ind_{d_1} - ind_{d_2}) \leq 0 \rightarrow |h_1 - h_2| \geq hwTime_k \\
&\forall(m_1, m_2, k) \in \mathcal{M}^2 \times \mathcal{K}\ s.t.\ side_{o_1} \neq side_{o_2}, \\
&(ind_{o_1} - ind_{d_2}) \times (ind_{d_1} - ind_{o_2}) \leq 0 \rightarrow |h_1 - h_2| \geq hwTime_k + hwTime_k
\end{aligned} \tag{4.1}$$

### 4.2.10 Objectives

The objective function $f$ is used to evaluate the quality of feasible solutions. It is a sum of individual costs which are:

1. Uncovered arrival/departure and unused initial trains cost ($f^{uncov}$),
2. Platform usage costs ($f^{plat}$),
3. Over-maintenance cost ($f^{over}$),
4. Train junction and disjunction operation costs ($f^{jun}$),
5. Non-satisfied preferred platform assignment cost ($f^{pref}$ ), and
6. Non-satisfied train reuse cost ($f^{reuse}$).

### 4.2.11 Uncovered arrivals/departures and unused initial trains

Minimizing the number of uncovered arrivals/departures and unused initial trains is important for the quality of the solution. Uncovered departures have no associated train in the solution schedule. Uncovered arrivals and unused initial trains have their associated trains not part of the solution schedule. The uncovered arrival/departure and unused initial train cost is given by:

$$f^{uncov} = uncovCost \times (|\{t \in \mathcal{T} \setminus \mathcal{T}^+\}| + |\{d \in \mathcal{D}; depTrain_d = \emptyset\}|) \tag{4.2}$$

If no train is assigned to a departure $d$ belonging to a joint-departure $j$, $d$ is considered uncovered. If trains are assigned to the other departures of $j$, these

departures are covered but their order has to be coherent with the order defined by the joint-departure. Likewise, if an arrival $a$ belonging to a joint-arrival $j$ is not covered, the trains associated with the other arrivals of $j$ (if they are covered) must be ordered consistently with the order defined by $j$.

### 4.2.12 Performance costs

#### 4.2.12.1 Platform usage costs

A platform may be used in four cases:

1. for an Arrival event only (let $\mathcal{A}^* \subset \mathcal{A}$ be the set of such arrivals),
2. for a Departure event only (let $\mathcal{D}^* \subset \mathcal{D}$ be the set of such departures),
3. for an Arrival event immediately followed in $sched_t$ by a Departure event ($\mathcal{Z} \subset \mathcal{A} \times \mathcal{D}$ represents the set of such arrival/departure pairs where $t$ does not leave the platform), or
4. for none of these cases.

Some costs apply to cases 1, 2 and 3; the associated cost functions are respectively denoted by $f^{plat_1}, f^{plat_2}, f^{plat_3}$, where $dwellCost$ represents the cost of one second of variation between the ideal stay duration and the actual stay duration on a platform. Note for any pair $(a, d) \in \mathcal{Z}$ we have $dwell_a = dwell_d$. For any arrival $a \in \mathcal{A}^*$, let $dwell_a$ be the duration of use of the platform assigned to $a$ (i.e. the time difference between the EnterResource and ExitResource events before and after $a$). Similarly, for any departure $d \in \mathcal{D}^*$, let $dwell_d$ be the duration of use of the platform assigned to $d$. And let $dwell_z$ and $idealDwell_z$ respectively denote the duration of use of a platform by a pair $z = (a, d) \in \mathcal{Z}$, and the ideal duration defined as the sum of ideal durations of $a$ and $d$:

$$idealDwell_z = idealDwell_a + idealDwell_d. \tag{4.3}$$

Then, we define:

$$f^{plat_1} = \sum_{a \in \mathcal{A}^*} dwellCost \times |dwell_a - idealDwell_a| \tag{4.4}$$

$$f^{plat_2} = \sum_{d \in \mathcal{D}^*} dwellCost \times |dwell_d - idealDwell_d| \tag{4.5}$$

$$f^{plat_3} = \sum_{z \in \mathcal{Z}} dwellCost \times |dwell_z - idealDwell_z| \tag{4.6}$$

Finally, we set:

$$f^{plat} = f^{plat_1} + f^{plat_2} + f^{plat_3}. \tag{4.7}$$

#### 4.2.12.2 Over-maintenance cost

Maintenance should be avoided when trains can still run for some time/distance because this generates additional production costs. Hence, for any maintenance operation, the remaining $DBM$ (expressed in seconds) or $TBM$ (expressed in km) of the concerned train is penalized. The corresponding costs, $remDCost$ and $remTCost$ are expressed per second and per km, respectively.

$$\begin{aligned} f^{maint} = & \sum_{e \in \mathcal{E}, y_e = BegMaintenance, c_e = "D"} remDCost \times remDBM_{t_e} + \\ & \sum_{e \in \mathcal{E}, y_e = BegMaintenance, c_e = "T"} remTCost \times remTBM_{t_e} \end{aligned} \tag{4.8}$$

#### 4.2.12.3 Train junction / disjunction operation cost

Each junction and disjunction operation has a cost in the objective function, $f^{jun}$ is the sum of these individual costs:

$$f^{jun} = \sum_{e \in \mathcal{E}, y_e = BegJunction} junCost + \sum_{e \in \mathcal{E}, y_e = BegDisjunction} disjCost \tag{4.9}$$

#### 4.2.12.4 Non-satisfied preferred platform assignment cost

For any arrival $a \in \mathcal{A}$, if the platform $platf_a$ assigned to $a$ is not in $prefPlat_a$, a cost of $platAsgCost$ applies. Likewise, this cost applies if, for any departure $d \in \mathcal{D}$, the platform $platf_d$ assigned to $d$ does not belong to $prefPlat_d$.

$$f^{pref} = \sum_{a \in \mathcal{A}, platf_a \notin prefPlat_a} platAsgCost + \sum_{d \in \mathcal{D}, platf_d \notin prefPlat_d} platAsgCost \tag{4.10}$$

#### 4.2.12.5 Non-satisfied train reuse cost

Finally, if some reuse $u \in \mathcal{U}$ is not satisfied, a cost of $reuseCost$ applies. $f^{reuse}$ is defined by:

$$f^{reuse} = \sum_{u \in \mathcal{U}, depTrain_{dep_u} \neq arrTrain_{arr_u}} reuseCost. \tag{4.11}$$

For the set of instances introduced for the competition, the first two objectives ($f^{uncov}$ and $f^{plat}$) are, by far, the most critical, as illustrated in figure 4.9.

## 4.3 Related Work

A large body of literature relative to train routing problems is available. However, any exact or even similar matches of previous research with the current problem could not be identified. Only variations to some of the sub-problems occurring here can be found in several publications (for example, in Lentink et al. [2003] and Freling et al. [2005]); moreover, a broad range of optimization models for specific problem variants does exist. We will not therefore be emphasizing any of the papers or related problem variants herein.

Recently, both during and after the competition, a few papers (or technical reports) have been published on this topic. Cambazard and Catusse [2014] propose a methodology heavily based on modeling with both Mixed Integer Programming (MIP) and Constraint Programming technologies for problem resolution. These authors mainly concentrate on solving the problem of assigning trains to departures and using a Mixed Integer Programming approach similar to that explained in this work. Haahr and Bull [2014] propose two exact methods, MIP and Column Generation, for solving the same sub-problem (called "Train Departure Matching Problem" in their paper). They report that solving the problem of assigning trains

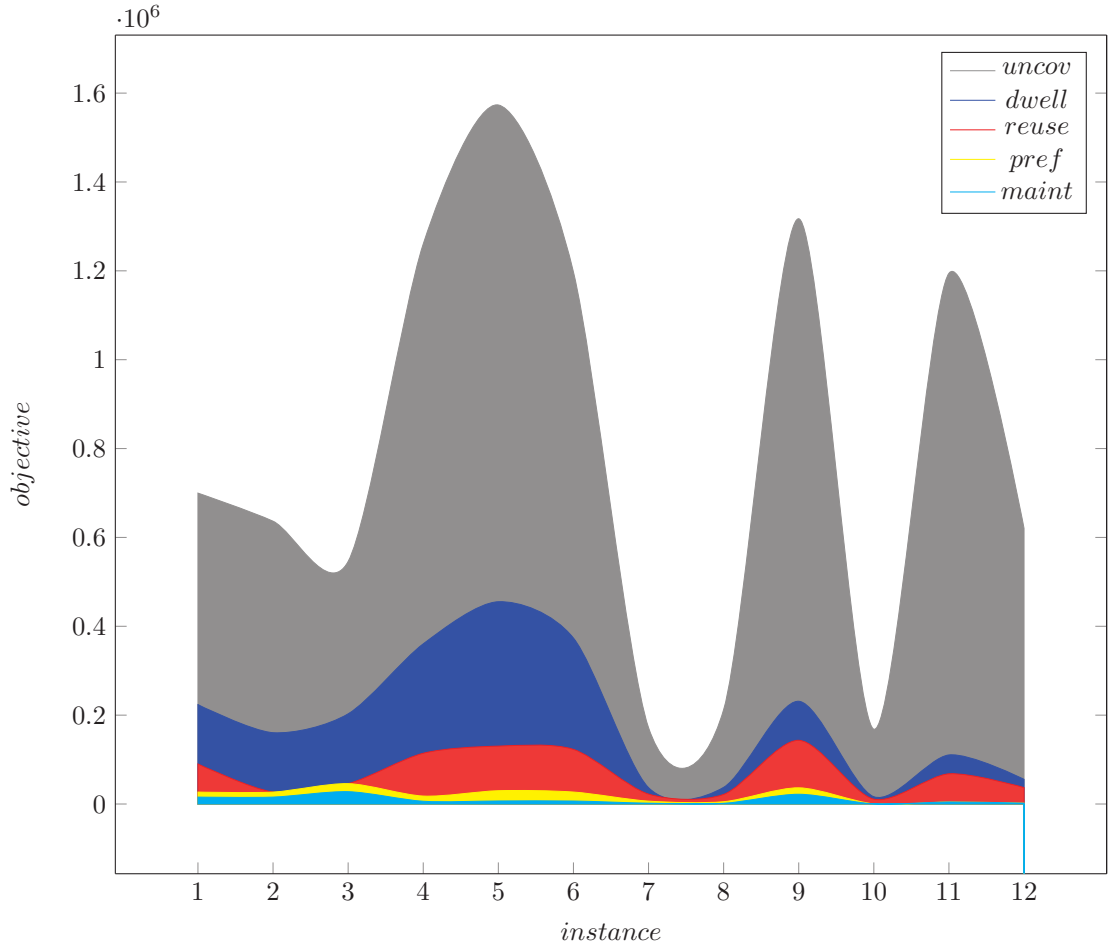


Figure 4.9: **Objective parts importance**: for each instance (B1 – B12), the value of each objective part in the final solution is represented by a different color.

to departures exactly is very difficult, if not impossible, for a given set of instances. Most of the teams competing in the ROADEF/EURO Challenge 2014 proposed algorithms that rely on greedy procedures or Integer Programming or a combination of both. Modeling an entire problem or a significant part of one using MIP is theoretically possible and has been achieved by a number of competitors, yet the outcome proved incapable of producing satisfactory results on the given set of instances. MIP techniques therefore are mainly used to solve only specific sub-problems. The decomposition of a problem into two dependent sub-problems, i.e. assignment and scheduling problems, is quite a natural step given the problem complexity and was carried out in most of the approaches presented. To the best

of our knowledge, a local search has not been conducted in any approach (at least as a significant component) besides the one presented herein.

## 4.4 Two Phase Approach

In our proposed method, the problem has been decomposed into two sub-problems, which are then solved sequentially. During the first phase, a train assignment problem (AP) is solved by combining a greedy heuristic and MIP. The main objective here is to maximize the number of assigned departures while respecting technical constraints. Other objectives are taken into account as well, with the aim of obtaining "better" input for the following phase. During the second phase, the train scheduling problem (SP), which consists of scheduling the trains inside the station, is solved using a constructive heuristic. The goal of SP is to schedule as many assignments as possible, in utilizing station resources and respecting all constraints. An iterative improvement procedure is implemented in order to improve the resulting schedule.

### 4.4.1 Simplifications

Several simplifications of the problem have been introduced in this work. Most of the work presented has been performed while competing in ROADEF/EURO Challenge, meaning that a reliable algorithm capable of producing feasible solutions for all instances within a limited computation time frame had to be developed. For this reason, when nearing the competition deadline, we decided to simplify our approach to a certain extent, namely by introducing some restrictions. The main restriction pertains to junction/disjunction operations for the trains. No such operation has been accepted in the final schedule. This restriction implies that joint trains should be set on exactly the same schedule, i.e. they must be scheduled at the joint departure or else one (or both) of them has to be cancelled. In both these cases, we can consider them as a single train in the scheduling phase of the algorithm. This same restriction applies to joint departures. The result may turn out to be, but not necessarily, slightly worse, yet on the other hand implementation of the scheduling procedure has been greatly simplified and become much more

reliable. Our numerical experiments have shown that this restriction does not exert a significant impact on the results obtained for the available set of instances. Another restriction consists of allowing only one maintenance per train. The underlying reasoning is the same, and similar conclusions can be drawn regarding the implementation and influence on final results. In the remainder of this chapter, it will be assumed that the restrictions described above have been applied.

### 4.4.2 Assignment problem

This section will describe the method adopted to solve the problem of matching (assigning) trains to departures. Assigning train $t \in \mathcal{T}$ to departure $d \in \mathcal{D}$ must satisfy the following technical constraints:

- compatibility: train category $cat_t$ must be compatible with departure $d$, i.e. $cat_t \in compCatDep_d$,
- the remaining distance/time before maintenance of train $t$ must be sufficient for departure $d$: $remDBM_t \geq reqD_d$ and $remTBM_t \geq reqT_d$,
- the time difference between arrival and departure must be large enough to allow executing required operations (train maintenance, changing direction/train reversal, ...),
- the number of maintenance operations per day constraint: the total number of maintenance operations during any day must not exceed a given number $maxMaint$.

We call an assignment $(t, d)$ of a train $t \in \mathcal{T}$ to a departure $d \in \mathcal{D}$ *feasible* if the following holds:

1. $cat_t$ is compatible with $d$, i.e. $cat_t \in compCatDep_d$,
2. $arrTime_t + minDwell(t, d) + maintTime(t, d) + addTime(t, d) \leq depTime_d$,
3. $remDBM_t \geq reqD_d, remTBM_t \geq reqT_d$,

where:

- $minDwell(t, d)$ is the minimum dwell time of train $t$, i.e. the minimum amount of time the train is to spend on the arrival/departure platform. This amount is equal to: $minResTime$ or $minRevTime$, depending on the set of possible platforms to choose,
- $maintTime(t, d)$ is the total maintenance duration required for scheduling $t$ to $d$ (0 if maintenance not required),
- $addTime(t, d)$ is an additional time necessary for parking and handling the train, *i.e.* in the case where the train is required to leave the arrivals platform before departure (non-immediate departure). The train may be parked either at the maintenance facility to undergo maintenance or at any authorized resource before being scheduled for departure.

Additional time, $addTime(t, d)$, is a variable value to be determined; it is used to increase the chance of finding a feasible schedule, yet an excessive value of this variable may also decrease the number of assigned departures. This value has been experimentally set to lie within the range of 5 to 60 minutes.
The following objectives are considered during the assignment phase:

1. (o1) maximize the number of assigned departures,
2. (o2) maximize the number of (possible) immediate departures [1],
3. (o3) minimize the number of maintenance operations,
4. (o4) minimize the number of assignments with a large difference between departure time and arrival time (called "long assignments") ($> 10h$, for example),
5. (o5) maximize the number of reuse assignments.

These objectives mixed and exact importance (weight) of each objective part will be given while describing the methods used for solving the problem. The reason

[1] Departure $d$ covered by train $arrTrain_a$ is said to be immediate if train $arrTrain_a$ can be scheduled to $d$ without leaving the platform. Such is the case if the time difference between arrival and departure, $depTime_d - arrTime_a$, does not exceed the maximum allowable dwell time, $maxDwell(a, d) = max(maxDwell_a, maxDwell_d)$.

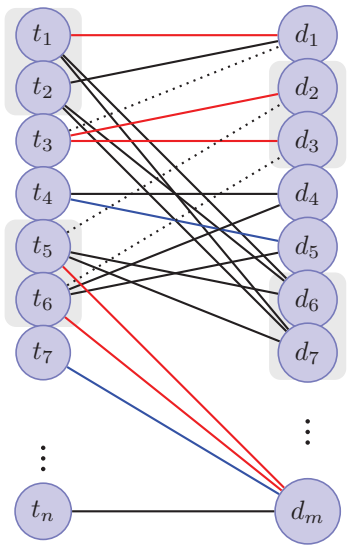


Figure 4.10: Assignment graph: color of the line corresponds to the number of required maintenances: black - 0 maintenances, red - 1 maintenance, blue - 2 maintenances. Dotted line represents link between arrivals and departures.

for introducing objectives (2) and (3) is to minimize the use of track groups since minimizing track group use will obviously decrease the chance of conflict. Another goal of inserting (3) is to minimize the use of maintenance facilities, which are considered critical resources. The aim in avoiding long waiting time between arrival and departure is to minimize the use of parking resources, especially yards. Violating the yard capacity constraint very often is the reason for cancelling an arrival or departure. This fact is based on numerical experimentation and our experience with the given set of instances. The final objective is expected to minimize the number of unsatisfied reuse assignments, though it was found to have a non-significant influence on the final results.

Other parts of the problem objective, such as platform usage cost, will only be considered during the scheduling phase.

Let's note that the dwell cost for an immediate assignment $(t, d)$, i.e. dwell cost of train $t$ scheduled to departure $d$ without leaving the platform, is known before the scheduling phase since time spent on the arrival/departure platform is known in advance (equal to $depTime_d - arrTime_a$).

The following definitions will be used herein:

- $nbM(t, d)$: the number of different maintenance types required to schedule train $t$ to departure $d$ (equals 0, 1 or 2);
- $imm(t, d)$: 1 if $d$ is immediate, 0 otherwise;
- $long(t, d)$: 1 if $depTime_d - arrTime_a > L$, where $L$ is a parameter, 0 otherwise;
- $reuse(t, d)$: 1 if reuse $(t, d)$ exists, 0 otherwise.

These same definitions apply when assigning a set of joint trains to a set of joint departures. One important remark to make is that if a set of $m$ joint trains $jt$ were assigned to a set of $m$ joint departures $jd$ without a disjunction operation and if one of the trains needed to undergo maintenance of type "D"("T"), then all the trains in $jt$ will undergo maintenance of this same type. The number of maintenance operations to be performed is therefore equal to: $m \times nbM(jt, jd)$.

The actual assignment problem difficulty depends on the structure of the considered instance. The existence of linked departures significantly complicates the AP since the remaining distance and time for some trains are not known before scheduling the linked departures. The problem also becomes more difficult if the maximum number of maintenance operations per day is low, i.e. the maximum number of maintenance operations per day constraint (MNMDC) is tight.
A combination of greedy and integer programming algorithms has been implemented to solve this assignment problem. A mixed integer programming approach could not be applied directly (independently) since the existence of linked departures makes the model uncontrollable.

#### 4.4.2.1 Greedy assignment algorithm

The first approach for solving an assignment problem is a greedy one, which consists of trying to match departures one by one. For each departure $d$, the best train is chosen in consideration of the defined objectives. In formal terms, the procedure works as follows:

- sort departures $d \in \mathcal{D}$ in ascending order with respect to departure time $depTime_d$;
- for each departure $d \in \mathcal{D}$, find the "best" available train.

Only feasible assignments are to be considered here and after assigning train $t$ to linked departure $d$, the data ($remDBM$, $remTBM$ and category) of the corresponding linked train are updated according to the constraint.
The exact choice of train for each departure is precisely described in the pseudo-code of the assignment algorithm, as given in 4.1. The "assignment value" for all possible feasible assignments is calculated and the train corresponding to the minimum value is chosen.
Informally, the following rules are applied when choosing the train for each departure $d$:

- consider only currently unassigned trains;
- whenever possible, always choose an immediate assignment: choose the one that minimizes dwell cost;
- assignments without required maintenance and trains with a small remaining distance/time value ($remDBM_t, remTBM_t$) are preferable for non-linked departures - the value $v = min(\frac{remD(t)}{reqD(d)}, \frac{remT(t)}{reqT(d)})$ is used as a "measure" of train size;
- in contrast to the previous rule, assignments involving required maintenance and trains with a large remaining distance/time value are preferable for linked departures;
- long assignments are not desirable (see objective (o4) defined in the previous section).

The maximum number of maintenance operations per day constraint is taken into account in the following manner. For each interval of days $[day_1, day_2]$, we define by $m(day_1, day_2)$ a current number of maintenance operations performed between

days $day_1$ and $day_2$. Obviously, $m(day_1, day_2)$ must not exceed $(day_2 - day_1 + 1) \times maxMaint$. Each time a maintenance operation or operations needs to be performed for assignment $(t, d)$, the value $m(day_1, day_2)$ is updated for each interval of days $[day_1, day_2]$ containing $[arrDay(t), depDay(d)]$. It will be shown below that respecting the given bound for each interval of days guarantees the existence of a feasible choice of days for each required maintenance operation in respecting the daily maintenance limit. A simple procedure for choosing the exact day of maintenance, along with the proof of correctness, is given in section 4.4.2.3. This same notion is found in the MIP model, which enables representing a constraint on the daily maintenance limit as linear.
The greedy procedure is combined with Integer Programming in order to improve the quality of assignments, as will be explained in the following section.

#### 4.4.2.2 Greedy assignment + MIP

To obtain improved solutions to the assignment problem, the greedy procedure explained above is combined with a Mixed Integer Programming (MIP) approach. The main difficulty in applying MIP directly (independently) to solve the assignment problem defined above pertains to the presence of linked departures. More specifically, the remaining distance and time before maintenance ($remDBM$, $remTBM$) of some trains is not known before assigning the linked departures. Deriving an efficient, solvable and complete MIP model able to take linked departures into account remains a challenge and the topic of future research. The greedy procedure described in the previous section and mixed-integer programming have thus been combined as follows:

- the assignment problem is solved by a greedy procedure;
- assignments of linked departures are fixed (in updating the data on linked trains);
- the resulting assignment problem is solved once again with MIP.

Let's define the set of possible assignments as:

**Algorithm 4.1:** Greedy assignment

```
1  Sort departures by time;
2  for d = 0 to |D| − 1 do
       // each departure
3      bestTrain ← −1;
4      minValue ← 10000000;
5      for t = 0 to |T| − 1 do
6          isFeasible ← d(t) =
           −1 ∧ isFeasible(t, d) ∧ checkNmbMaintenancesConstraints();
7          if isFeasible then
8              value ← 10000000;
9              if LONG then
10                 value ← depTime_d − arrTime_t;
11             else
12                 if IMMEDIATE then
13                     value ← −100000 + dwellCost(t, d);
14                 else
15                     nmbM ← calculateNmbRequiredMaintenances(t, d)
                       value = min(remD(t)/reqD(d), remT(t)/reqT(d))
16                     if d is not linked and nmbM > 0 then
17                         value ← value + nmbM ∗ 100000 ;
18                     if d is linked and nmbM > 0 then
19                         value ← value − 100000;
20                     if d is linked and nmbM = 0 then
21                         value ← −value
22             if value < minValue then
23                 minValue ← value;
24                 bestTrain ← t;
25     if bestTrain ≠ −1 then
26         t(d) ← bestTrain;
27         d(bestTrain) ← d;
28         Update linked arrival if needed;
29         Add maint. to all intervals containing [arrDay, depDay] if required;
```

$$\mathcal{S} = \{(TR, DEP) : TR \subset \mathcal{T}, DEP \subset \mathcal{D}\} = \{(T_1, D_1), (T_2, D_2), \ldots, (T_n, D_n)\},$$

where $TR$ and $DEP$ are the sets of arrivals and departures respectively and:

- all trains (departures) in $TR$ ($DEP$) are joint (this assumption is considered true if the set contains just one element, i.e. $|TR| = 1$ ($|DEP| = 1$));
- the number of trains in $TR$ equals the number of departures in $DEP$, i.e. $|TR| = |DEP|$;
- all assignments are feasible (the definition of feasibility for joint assignments is analogous to that for single assignments);
- linked assignments do not exist, i.e. $DEP$ is not a linked departure.

For example, given two joint arrivals $t_1$, $t_2$ and two joint departures $d_1$, $d_2$, then: $\mathcal{S} = \{(t_1, d_1), (t_1, d_2), (t_2, d_1), (t_2, d_2), (\{t_1, t2\}, \{d_1, d2\})\}$.

For each pair in $S$, a decision variable is defined. More formally, the binary variable $x_j$ corresponds to the $j^{th}$ candidate assignment $(T_j, D_j)$; for each of $n$ possible assignments ($1 \leq j \leq n$), we obtain $x_j = 1$ if $T_j$ is assigned to $D_j$, $x_j = 0$ otherwise. Assignments of linked departures are previously fixed, and the variables pertaining to fixed trains and departures have not been included herein.

As mentioned above, several objectives must be taken into account during the assignment phase ((o1)-(o5)). All these objectives are merged into a single objective function by applying a weight for each of the parts. Formally, the MIP objective function is to maximize the following weighted sum:

$$\begin{aligned}\sum_{j=1}^{n} |T_j| \times x_j \times ( \quad & assignmentWeight + \\ & durationWeight \times (1 - long(T_j, D_j)) + \\ & immWeight \times imm(T_j, D_j) + \\ & maintWeight \times (2 - nbM(T_j, D_j)) + \\ & reuseWeight \times reuse(T_j, D_j)),\end{aligned}$$

with all weights being non-negative and ordered by magnitude. The weights are chosen experimentally, and all results in this chapter have been obtained with the following choices: $assignmentWeight = 1000$, $durationWeight = 100$, $immWeight = 10$, $maintWeight = 1$, $reuseWeight = 1$.

Next, let's define the constraints included in this model.
Let $T_j()$ (*resp.* $D_j()$) denote the characteristic function of $T_j$ (*resp.* $D_j$): $T_j(t) = 1$ if train $t$ belongs to the set $T_j$, $T_j(t) = 0$ otherwise. A maximum single assignment exists for each train $t$ and each departure $d$, as expressed by the following constraints:

$$\forall t \in T \quad \sum_{j=1}^{n} T_j(t) \times x_j \leq 1$$
$$\forall d \in D \quad \sum_{j=1}^{n} D_j(d) \times x_j \leq 1$$

As mentioned above, train disjunctions and junctions are prohibited, and this restriction must therefore be included in the model. This rule means that if two trains $t_1$ and $t_2$ belonging to a joint arrival $ja$ are assigned to departures $d_1$ and $d_2$, then $d_1$ and $d_2$ must belong to the same joint departure $jd$, and vice versa. This constraint can be simply included in the model by the following. Let $(TR, DEP) \in \mathcal{S}$ be a candidate assignment, such that $|TR| > 1$. The disjunction of trains belonging to $TR$ is prohibited by allowing at most one variable corresponding to $(tr, dep)$, such that $tr \subseteq TR$ equals 1. A similar constraint is used to prohibit train junctions. Formally, these constraints can be expressed as follows:

$$\forall TR : |TR| > 1 \wedge (\exists DEP : (TR, DEP) \in \mathcal{S}) \quad \sum_{(T_j, D_j) \in \mathcal{S}, T_j \subseteq TR} x_j \leq 1$$
$$\forall DEP : |DEP| > 1 \wedge (\exists TR : (TR, DEP) \in \mathcal{S}) \quad \sum_{(T_j, D_j) \in \mathcal{S}, D_j \subseteq DEP} x_j \leq 1$$

Let $M_j = |T_j| \times nbM(T_j, D_j)$ denote the number of maintenance operations required by the assignment $(T_j, D_j)$. The first and last possible days for maintenance (as denoted by $fd_j$ and $ld_j$) are the corresponding arrival and departure days. For each interval $[d_1, d_2]$ $(1 \leq d_1 \leq d_2 \leq nbDays)$, the right-hand side values are initialized: $MaxMaint(d_1, d_2) = (d_2 - d_1 + 1) \times maxMaint$. Then, for each previously

fixed assignment $(t, d)$ (linked assignments), $MaxMaint(d_1, d_2)$ is decreased by the number of maintenance operation(s) required by $(t, d)$ for all $[d_1, d_2]$, such that $[arrDay(t), depDay(d)] \subseteq [d_1, d_2]$. The constraint on maintenance is formulated by the following:

$$\forall [d_1, d_2] \quad \sum_{j=1}^{n} M_j \times x_j \times \mathbb{1}_{[fd_j, ld_j] \subseteq [d1, d2]} \leq MaxMaint(d_1, d_2)$$

The assignment values obtained by the greedy procedure and MIP are illustrated in Table 4.1. The three most critical values (in our experiment) are reported as: the number of unassigned departures, the number of immediate departures, and the number of long assignments.

| Inst | Greedy | | | Greedy+MIP | | |
|---|---|---|---|---|---|---|
| | #nonassign | #immediate | #long | #nonassign | #immediate | #long |
| B1 | **138** | **350** | 202 | 143 | 339 | **159** |
| B2 | **138** | **350** | 202 | 143 | 339 | **159** |
| B3 | 131 | **319** | 272 | **109** | 284 | **173** |
| B4 | 155 | **602** | 230 | **136** | 576 | **207** |
| B5 | 180 | **702** | 283 | **153** | 648 | **269** |
| B6 | 150 | **603** | 230 | **131** | 576 | **209** |
| B7 | **31** | 128 | 13 | 32 | **134** | **7** |
| B8 | **33** | 124 | 13 | 34 | **131** | **7** |
| B9 | **116** | 741 | 359 | 119 | **742** | **251** |
| B10 | 48 | 39 | 4 | **48** | **40** | **0** |
| B11 | 274 | 199 | 219 | **253** | **205** | **218** |
| B12 | 135 | 126 | 72 | **132** | **128** | **63** |

Table 4.1: **Assignment values B:** number of non-assigned departures, number of immediate departures and number of "long" assignments ($depTime - arrTime > 10hours$) is given for each instance and each method used. $addTime = 30\ minutes$ is used.

#### 4.4.2.3 Choosing maintenance days

As described earlier, maximum number of maintenances in a day constraint is satisfied by respecting the limit for each interval of days $[d1, d2]$, while updating the number of maintenances for each interval that contains $[arrDay(t), depDay(t)]$ when maintenance has to be done for train $t \in \mathcal{T}$. Exact day for each maintenance remains to be determined. Two simple procedures, along with the proofs of correctness, for choosing the maintenance days used in our experiments are presented below.
The first procedure works in the following way:

- sort assignments (that require maintenance) in ascending order by departure day and then by arrival day. Sorting example is given in Figure 4.11.
- for each assignment $(t_i, d_i)$ in a sorted list
  - choose the first (minimum) available day for maintenance, i.e. the first day in $\{arrDay(t_i), \ldots, depDay(t_i)\}$ for which maintenance limit $maxMaint$ is not reached.

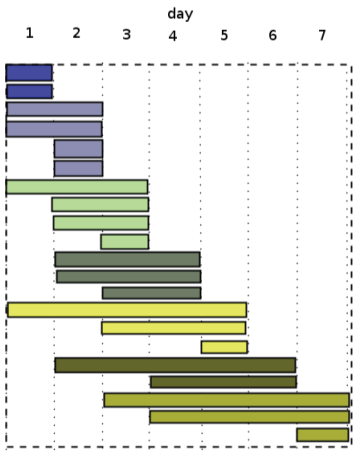


Figure 4.11: Sorting Example

Now, we will prove the correctness of this procedure.
Let $M = \{m_1, m_2, ..., m_k\}$ be the set of all assignments that need maintenance. We

can assume that only one maintenance is required for each assignment (assignment that requires 2 maintenances can be represented as two assignments that require a single maintenance). Let $a_i$ and $d_i$ be arrival and departure day of assignment $m_i$ respectively. We will write $m_i = (a_i, d_i)$, $a_i, d_i \in \{1, 2, \ldots, nbDays\}$.

**Claim**: If the following inequality holds for each interval of days $[d_1, d_2]$:

$$\sum_{i=1}^{k} \mathbb{1}_{[a_i,d_i]\subseteq[d1,d2]} \leq (d_2 - d_1 + 1) * maxMaint \tag{4.12}$$

then assignment of maintenance days using the procedure described respects maximum number of maintenances in a day constraint.

We will prove the claim by contradiction. Let $m_j = (a_j, d_j)$ be the first assignment for which maintenance day cannot be chosen and $M_{j-1} = \{m_1, m_2, \ldots, m_{j-1}\}$ be the set of previously assigned candidates.
This means that maintenances limit is reached for each day in $[a_j, d_j]$. (*)
Let $d_0$ be a minimum (first) day such that maintenance limit is reached for each day in interval $[d_0, d_j]$ ( existence of $d_0$ is obvious because of (*) ).
Obviously, $d_0 = 1$ or limit is not reached for $d_0 - 1$. (**)
Let $F = \{f_1, f_2, \ldots, f_l\} \subset M_{j-1}$ be the set of assignments already assigned to one of the days in $[d_0, d_j]$ ($l = (d_j - d_0 + 1) \times maxMaint$ since limit is reached for the whole interval).
For each $f_i = (a_{fi}, d_{fi}) \in F$ we have

- $a_{fi} \geq d_0$ : otherwise, chosen maintenance day would be not greater than $d_0 - 1$ because of assignment rule and (**)
- $d_{fi} \leq d_j$ : because of sorting order

The same holds for $m_j$ : $a_j \geq d_0$ because of definition of $d_0$ and (*), $d_j \leq d_j$.

Now, consider the set $U = F \cup m_j$. As shown, maintenance interval for each element of $U$ lies in $[d_0, d_j]$ and thus,

$$\sum_{i=1}^{k} \mathbb{1}_{[a_i,d_i]\subseteq[d_0,d_j]} \geq \sum_{i\in U} \mathbb{1}_{[a_i,d_i]\subseteq[d_0,d_j]} = |U| = (d_j - d_0 + 1) \times maxMaint + 1$$

which is contradictory to the main assumption (1).

The second procedure for choosing maintenance days consists of repeating the following two steps:

- STEP1: randomly choose a maintenance $m_j$ that has not been fixed yet
  - maintenance $m_j$ has not been fixed if $a_j < d_j$
  - if such maintenance does not exist, assignment of days to maintenances is finished and procedure terminates
- STEP2: fix maintenance day and update the constraints (system of inequalities (1))
  - randomly choose a maintenance day $x_j$ in interval $[a_j, d_j]$ such that fixing a day of maintenance $m_j$ to $x_j$ (reducing the domain of variable) and updating all necessary constraints will not violate any constraints (system of constraints/inequalities remains feasible)

It is clear that described procedure will give a feasible choice of days for maintenances if STEP2 can be performed at any iteration. Checking if fixing a day of maintenance $m_j$ to $x_j \in [a_j, d_j]$ will keep the system (1) feasible is straightforward:

- increase the left hand side in each constraint from (1) if it corresponds to interval of days $I$ such that $[a_j, d_j]$ is not contained in $I$ and $x_j$ is contained in $I$;
- if some of the constraints (updated in previous step) becomes violated, then $x_j$ is not a feasible choice for maintenance $m_j$ and we need to choose a different day to test.

Note that left hand side of any constraint can be increased maximum by 1 when fixing a single maintenance.

Now, we will prove that STEP2 of the procedure for choosing maintenance days can be performed at any iteration. Let $m_i = [a, d]$ be any non-fixed maintenance i.e. $a < d$. It is enough to prove that domain $[a, d]$ can be reduced to a smaller one without violating any constraints. Then, by induction (or by repeating the process), interval will be reduced to a single day. Specifically, domain $[a, d]$ can be reduced to $[a, d-1]$ or $[a+1, d]$. Assume the contrary, i.e. $[a, d]$ cannot be reduced to any of two intervals $[a, d-1]$, $[a+1, d]$. This implies that there are two intervals $I_1 \supseteq [a, d-1]$ and $I_2 \supseteq [a+1, d]$ such that corresponding constraints are active (equalities hold in both inequalities from (1) corresponding to intervals $I_1$ and $I_2$). Constraints/inequalities affected by reducing the domain to $[a, d-1]$ are the ones that correspond to the intervals that contain $[a, d-1]$ and do not contain $[a, d]$, i.e $[a, d-1], [a-1, d-1], \ldots, [1, d-1]$. Thus, $I_1 = [x, d-1]$ for some $x \in \{1, 2, \ldots, a\}$. Similarly, we have $I_2 = [a+1, y]$ for some $y \in \{d, d+1, \ldots, nbDays\}$. Equality in constraint corresponding to $I_1 = [x, d-1]$ means that there are exactly $(d-x) \times maxMaint$ maintenances (intervals) inside $I_1$ (number of ones in left hand side of constraint). Similarly, there are $(y-a) \times maxMaint$ intervals inside $I_2$. We will denote by $n(I)$ the number of maintenances contained in interval $I$ (corresponding interval of the maintenance is contained in $I$).
Clearly, the number of intervals contained in either $I_1$ or $I_2$ is not greater than the number intervals contained in $[x, y]$, since $I_1, I_2 \subseteq [x, y]$. Formally,
$n([x, y]) \geq n(I_1) + n(I_2) - n(I_1 \cap I_2))$.
Also, interval $[a, d]$ is included in $[x, y]$ and not included in any of $I_1, I_2$ and $[a, d]$ is the interval corresponding to maintenance $m_i$ so we have stronger inequality:
$n([x, y]) \geq n(I_1) + n(I_2) - n(I_1 \cap I_2)) + 1$. i.e.
$n([x, y]) \geq n(I_1) + n(I_2) - n([a+1, d-1])) + 1$
Constraint corresponding to $[a+1, d-1]$ in (1) is $n([a+1, d-1]) \leq (d-a-1) \times maxMaint$ (note that inequality also holds if $a+1 = d$ i.e. when intervals have no intersection), so we have

$$\begin{aligned} n([x,y]) &\geq n(I_1) + n(I_2) - (d - a - 1) \times maxMaint + 1 \\ &= (d - x) \times maxMaint + (y - a) \times maxMaint \\ &- (d - a - 1) \times maxMaint + 1 \\ &= (y - x + 1) \times maxMaint + 1 \end{aligned} \tag{4.13}$$

The last inequality is contradictory to the inequality in (1) corresponding to interval $[x, y]$.

### 4.4.3 Scheduling problem

The goal of the second algorithm part is to schedule the assignments generated by the first phase inside the station while respecting all resource constraints. Trains must move through the network/graph of inter-connected resources. All types of resources and constraints associated with the trains are given in the problem description provided in Section 4.2. A constructive procedure has been implemented here to solve the scheduling problem. The output schedule is then improved by an iterative procedure based on a local search.

Three possibilities exist for the schedule of each train $t \in \mathcal{T}$:

1. $t$ is scheduled to departure $d \in \mathcal{D}$;
2. $t$ is parked inside the station until the end of the planning time frame without being assigned to any departure;
3. $t$ is cancelled.

The schedule, possibly an empty one, must be given for each train $t \in \mathcal{T}$. All resources used by the train must be specified, along with the exact time of entering and leaving each resource. The greedy procedure schedules the trains one by one, in a defined order (ordering will be addressed in Section 4.4.3.10). A complete schedule for the train is output before scheduling the next train. Nevertheless, all trains share the same resources and all constraints need to be respected over the entire scheduling procedure. If part of the train schedule can only be generated by violating one or more constraints, then the train is cancelled, i.e. it will have

an empty schedule.

The following strategy for parking the trains and performing maintenance is employed. Maintenance resources are considered critical in this problem and therefore yards are used as a parking resource whenever possible. Maintenance facilities are used for parking only when parking on the yard is impossible. Similarly, single tracks are never used for parking the trains, rather they are merely used as transition resources. If maintenance is required for a particular train, it is scheduled as quickly as possible. If the maintenance facility cannot be found immediately after arrival or if maintenance has to be performed on another day (see Section 4.4.2.3), the train is parked at the yard and moved to the maintenance facility as soon as possible. After performing a maintenance operation, the train can stay at the facility and move to a later departure (provided the waiting time is not too long) or else it should be moved to the yard as soon as possible.

Since junction and disjunction operations are prohibited, a set of joint trains (departures) can be considered as a single train (departure). If one of the trains belonging to a joint arrival is assigned (during the assignment phase) to a departure while another one is not, then the other train will have an empty schedule, which naturally would be penalized in the objective function. This same set of rules is applied to joint departures. In the remainder of this section, it will be assumed that all trains and departures are single. Nevertheless, a set of joint trains still contains two or more trains and all the costs relative to this "joint train" are multiplied, as is resource consumption (except in the case of track groups where moving a joint train is considered as a single move).

#### 4.4.3.1 Possible train movements

Modeling the scheduling problem exactly, i.e. in considering all possible resource choices (all possible train movements) at every possible instant, is not realistic given the size and structure of the instances proposed by SNCF. We have therefore limited possible train movements to the following:

- arrival - arriving on the platform via a given set of track groups (arrival

sequence),

- departure - departing from the platform via a given set of track groups (departure sequence),
- move from arrivals platform to yard,
- move from arrivals platform to facility,
- move from parking (facility, yard, single track) to departure platform,
- move from yard to facility,
- move from facility to yard.

The train schedule will specify, for each train movement, the set of resources deployed with the exact resource input (output) times. The connected set of resources used while moving the train from one place to another will be called *path*. $P = (R_1, R_2, \ldots, R_k)$ denotes a path connecting resources $R_1$ and $R_k$ that starts at $R_1$, visits resources $R_2, R_3, \ldots, R_{k-1}$ and then ends at $R_k$. Two consecutive resources in a path must be connected by a gate. The use of path $P$ (i.e. using resources in $P$) for a given entry and exit times on each resource will be called *travel*. To simplify the scheduling procedure, let's assume that the time spent on each intermediate resource on a path (resources $R_2, \ldots, R_{k-1}$) is always a minimum, i.e. no waiting on any intermediate resource once the minimum time has elapsed. In the case of track groups, this time is set equal to $trTime$, while in the case of other resources it equals $minResTime$ or $minRevTime$, depending on the resource input and output sides. Consequently, a travel duration is known and equal to the sum of minimum resource times for each intermediate resource on the path.
The travel of a given train is thus determined by both the designated path and the travel starting time. Travel using path $P$ and starting at time $st$ will be denoted $T(P, st)$.
To conclude, the schedule of each train is represented as a set of travels, hence the decision variables to be determined for each scheduled train are a set of paths and

starting times.
All paths potentially used for any feasible movement are constructed before the start of the scheduling procedure. This set of paths includes those for each pair of resources $(r_1, r_2)$, such that $r_1$ and $r_2$ are of different types, with neither of them being a track group or single track. The simple and common depth-first search (DFS) algorithm serves to identify all these paths. This simple preprocessing step simplifies implementation to a significant extent. Paths are sorted by length (i.e. total number of resources) for each pair $r_1, r_2$; moreover, should many paths exist between two resources, only several shortest ones are to be kept.

#### 4.4.3.2 General rules for choice of movements

As mentioned earlier, the set of possible train movements is limited to the simplest and most significant ones. Furthermore, the scheduling procedure seeks to identify a feasible schedule for a given train with lowest possible number of movements, i.e. unnecessary movements should not be performed.
The choice of movements depends on the type of operations that need to be carried out (e.g. maintenance), total time spent at the station, *etc.* The schedule for the given train is built by scheduling each travel one at a time. In some cases, two travels need to be scheduled simultaneously during the same procedure.
The general strategy for choosing the movements of train $t$, corresponding to arrival $a$, may be summarized as follows:

1. Immediate departure: if the time between arrival $a$ and departure $d$ assigned to $t$ is short enough. The train will only use track group resources to arrive at the platform, with a platform stay from $arrTime_a$ to $depTime_d$, and then track group resources to depart from the platform (i.e. to exit the system). The task is to determine the gates on the track groups that avoid conflicts, as well as find a platform available between $arrTime_a$ and $depTime_d$ that respects both the train order and platform length constraints.

2. Park on any yard: if the time between arrival and departure is sufficiently long and no maintenance is required or feasible travel to the maintenance facility could not be found. This task is to find an available yard, feasible

travel to the yard (resources and gates without conflict) and an available arrivals platform.

3. Go to maintenance: if maintenance is required. This task is to find an available maintenance facility, feasible travel to the facility and an available arrivals platform.

If the train is currently parked (yard, maintenance facility), the following options become available:

4. Move from yard to departure: if the train is positioned at the yard and needs to be scheduled for departure without first performing the maintenance. Shortly before departure, the train will be moved to the departure platform and depart at a given time. This task is very similar to (2), but in the other direction.

5. Move from yard to maintenance facility and perform maintenance: if the train is positioned at the yard and needs to be scheduled for departure after performing maintenance. Movement should be made as soon as possible.

6. Move from maintenance facility to departure: if the train is positioned at the maintenance facility and needs to be scheduled for departure.

7. Move from maintenance facility to yard: if the train positioned at the maintenance facility needs to be scheduled for departure and the departure time is not "close".

#### 4.4.3.3 Resource consumption and travel feasibility

The consumption of each resource in the station needs to be updated during the schedule for each train and the resource availability (i.e. resource constraints) must be checked. Resource consumption is tracked by recording the set of all previous visits for each resource in the station. The set of visits to the resource is updated when scheduling each train. A visit to a resource has the following attributes: entry time and side, exit time and side, number of trains (1,2,...), length of trains, and entry and exit gates. Each time the resource needs to be visited by a train, all

constraints for a given resource are checked and the visit is only allowed if found to be feasible.
Then, verifying the feasibility of travel $T(P, st)$ simply requires checking the visit feasibility on each resource in path P.

#### 4.4.3.4 Time spent on a resource between two travels

One of the difficulties involved in train scheduling is to determine an exit time for the last resource in each travel/path. Knowing the exit time on the last resource is required in order to check constraints regarding this particular resource. An exact exit time is often not known before the next travel has been planned. The following strategy has been employed to handle this issue:

1. If the last resource of travel $i$ is a platform (in the case of arrival and departure): travels $i$ and $i+1$ must be planned together. This step is equivalent to planning a single travel, with possible paths being a combination of two paths (candidate paths for travels $i$ and $i+1$), yet time spent on the platform is no longer fixed and needs to be determined.

2. If the last resource of travel $i$ is a yard (for parking): the exit time is equal to departure time if the train must be scheduled for a departure; otherwise, the exit time is the end of the time horizon. A train parked on a yard that cannot be scheduled for a later departure is cancelled.

3. If the last resource of travel $i$ is a facility (for either parking or maintenance)

   - the same as (2) in the case where the facility is only used for parking
   - exit time is equal to $\max(depTime_d, enter\_time + max\_fac\_time)$ if the train is assigned to departure $d$ and maintenance needs to be performed, where $enter\_time$ is the time at which the train enters the facility and $max\_fac\_time$ is a parameter defining the maximum length of time the train can stay at the facility.

If a feasible travel cannot be found (e.g. for moving from yard to facility), then the train will be cancelled, i.e. no attempt will be made to change the previous

travels.

#### 4.4.3.5 Travel starting time

An important decision to be made when scheduling each train is the starting time for each travel. Some starting times are fixed, such as the time of arrival and departure, while others are to be selected from a feasible set of time instants. An ideal starting time will be defined for each travel and it will be attempted to schedule travel with a starting time as close as possible to the ideal time. We can always calculate the earliest and latest possible travel times, $est$ and $lst$, which depend on the time constraints such as minimum resource times, travel duration, fixed arrival and departure times, etc.
The ideal travel starting time depends on the type of travel; for our purposes, the following was used:

- If train $t$, corresponding to arrival $a$, needs to be moved from platform to parking (yard or facility): the ideal starting time will minimize the dwell cost on the platform, i.e. $arrTime_t + idealDwell_a$;

- If train $t$, parked at a yard or facility, needs to be moved to the platform for departure $d$: the ideal starting time will minimize the dwell cost on the platform, i.e. $depTime_d - idealDwell_d - travelDur$, where $travelDur$ is the duration of travel to the platform;

- If the train is to be moved from one parking resource to another (i.e. from yard to facility and vice versa): the ideal starting time is the earliest possible starting time, $est$

Once the ideal travel starting time, $idealST$, has been determined, the next step seeks to choose a starting time, between the earliest and latest possible, as close as possible to $idealST$. Formally speaking, the selected starting travel time, $st$, is the first one from the set

$$\{idealST, idealST - \delta, idealST + \delta, idealST - 2\delta, idealST + 2\delta, \ldots\},$$

such that $st \in [est, lst]$ and travel $T(P, st)$ is feasible for some path $P$. Slick time, $\delta$, is chosen from the interval $[10s, 60s]$.

#### 4.4.3.6 Choosing platforms and parking resources

Each arrival and departure is assigned to the platform that minimizes the sum of dwell cost and preferred platform cost. Parking resources (yards and facilities) and paths associated with these resources are sorted by "path length", and the first feasible path is chosen.

#### 4.4.3.7 Dealing with yard capacity

As mentioned earlier, the main resources used for train parking are yards. Each yard has a capacity that cannot be exceeded at any time. The numerical experimentation on a given set of instances shows that yards are critical and scarce resources for this scheduling problem. A strategy must therefore be developed to make better use of the yards. Since the station has limited capacity, it is not possible for the number of trains arriving at the station to significantly exceed the number of departures from the station. Consequently, most trains associated with arrivals must be scheduled to a departure. However, some trains may remain at the station until the end of the planning horizon, though this number is typically much smaller than the number of trains scheduled to a departure. Furthermore, if station resources are critical, especially yards, it is not desirable to consume them with the trains not scheduled to any departure, which potentially could disable the scheduling of some trains to be scheduled for departure. We have therefore used the following simple heuristic in the scheduling procedure:

1. *planning departures*: schedule each assigned train $t$ ($d(t) \geq 0$) and if the train cannot be successfully scheduled for departure $d(t)$, then cancel it;

2. *park unassigned and cancelled trains* at the very end of the procedure (after optimizing the solution) - in respecting capacity constraints.

Let's also note that assignments with too much time between arrival and departure are not desirable from the standpoint of yard capacity, which is taken into account during the assignment phase.

#### 4.4.3.8 Choosing gates: Avoiding conflicts on track groups

The main difficulty with this problem, from our experience, lies in effectively choosing the gates for each track group to enter and exit, as this gate selection will allow more trains to travel on the track groups without conflict. As defined in Section 4.2, conflicts on track groups are prohibited. For each travel $T(P, st)$ of train $t$, a set of entry/exit gates on each track group in $P$ needs to be determined. Like for all other resources, the exact entry and exit times for each track group are known if the starting time, $st$, of the travel is given.

Let $n_1$ be the number of possible gates to choose for an entering track group $TG \in P$, and $n_2$ the number of possible gates for exiting; we then have a total of $n_1 \times n_2$ possible moves to choose from. It is simple to check whether or not the selected move conflicts with any of the previous moves on the track group. For this purpose, like for any other resource type, we have kept a set of all visits (moves) to the track group, and only those moves not in conflict with any moves in the given set are to be allowed.

Since the number of moves on the same track group can often be large, considering all moves when detecting potential conflicts can be very time consuming. We have therefore grouped all visits to the track group into subsets, determined by entry time, which then allows conflicts to be detected by considering just a few visit subsets.

Formally, for track group $k$, $m = |\mathcal{H}|/(trTime_k + hwTime_k) + 1$ subsets $S_1$, $S_2$, $\ldots$, $S_m$ are created, with subset $S_i$ containing all visits with an entry time in $[(i-1) \times (trTime_k + hwTime_k), i \times (trTime_k + hwTime_k)]$. When potential conflicts need to be detected for a visit with entry time $eT$, only three subsets, $S_{j-1}, S_j, S_{j+1}$ require consideration, where $j = eT/((trTime_k + hwTime_k))$. The number of moves in a single subset is usually very low, which tremendously accelerates the conflict detection procedure.

A set of entry/exit gates without conflicts must be determined for the entire travel

$T(P, st)$, which means that a feasible move needs to be found on each track group in $P$. Consequently, the number of possible combinations of moves becomes greater. It has been assumed here that path $P$ and travel starting time $st$ are both known.

A simple DFS procedure to find a feasible set of moves for travel has been employed. This procedure explores all possible combinations of moves (one move for each track group in $P$) until a feasible one (without conflicts) has been found.
The naive way of using a DFS procedure is to begin with the first possible gate on each track group and increase the gate index, according to a depth-first sequence, whenever a feasible choice has not been found. This manner of choosing the gates is not necessarily a good one as regards track group usage. To improve the choice of gates, let's attempt to identify a different order for exploring the possibilities in a DFS procedure. Formally, for each path $P = (R_1, R_2, \ldots, R_n)$, a "preferred" entry gate on each resource in $P$ will be defined and the DFS procedure will explore all possibilities by starting with a preferred gate on each resource. Note that the preferred gate is fixed when only one gate exists, as in the case of individual resources.
The set of preferred gates for travel $T(P, st)$ is determined according to the first and last resources, more specifically $R_1$ and $R_n$ in $P$, and depending on the positions of these resources relative to the neighboring track groups, $R_2$ and $R_{n-1}$.
It can be noted that the majority of travels start or end at the platform, i.e. they have a platform as the first or last resource. Such is actually the case for all travels in our set-up, except for movements from yard to facility and vice versa. Consequently, the most critical track groups are those either connected to or close to the platforms. We have therefore decided to define the preferred gates solely according to the relative position of the platform with respect to the connected track group. If $R_1$ is the $j - th$ of $np$ platforms connected to track group $R_2$ (according to gate indices) and $g_1, g_2, \ldots, g_k$ are the gates from $R_i$ to $R_{i+1}$ $(2 \leq i < n)$, then a preferred gate is $g_l$, where: $l = \frac{j}{np}k$. The same rule is applied if $R_n$ is a platform. For example, if the chosen platform is the top platform, then it is only natural to choose the top gate on each resource in path P.
We have conducted several experiments with a more complicated choice of gates, however the results obtained only changed slightly and were not necessarily always

better. Moreover, the local search procedure described at the end of this section will question this choice of gates.

#### 4.4.3.9 Virtual visits

One of the difficulties in avoiding conflicts is not knowing the "future traffic", i.e. overall track group use. This issue is especially important when choosing the starting times of travels without a fixed starting time (i.e. all travels except arrivals and departures). For example, if train $t$, associated with arrival $a$, needs to be moved from the arrivals platform to the yard, a possible starting time for moving to the yard would lie in the interval $[arrTime_a + minResTime, arrTime_a + maxDwell_a]$. Very often, many different possibilities are feasible and just one has to be chosen, although choosing any one of them might potentially block more trains yet to be scheduled than choosing another one.
We have introduced the concept of "virtual visits" to improve the starting time of each such travel. Virtual visits can be viewed as the potential visits capable of occurring on the track groups in the future. Virtual visits will be generated for each arrival and each matched departure (by the assignment procedure) and then taken into account when choosing the starting times and gates for the travel.
The set of virtual visits $V$ is constructed as follows:

- for each arrival $a \in \mathcal{A}$ and each matched departure $d \in \mathcal{D}$
  - randomly choose a compatible platform $p$,
  - find a set of gates for $arrSeq_a \cup p$ ($p \cup depSeq_d$) with a minimum number of conflicts with $V$, in applying the procedure explained in the previous section,
  - add the corresponding set of visits to the track groups to $V$.

The set of virtual visits is computed at the start of the scheduling phase, before scheduling any train. Next, during train scheduling, the starting time of each travel not corresponding to an arrival or departure is selected in order to minimize the number of conflicts with virtual visits. The virtual visits of train $t$ are removed

from $V$ when the scheduling procedure for $t$ has been completed ($t$ is scheduled for departure, parked or cancelled).

#### 4.4.3.10 Scheduling order

Trains are to be scheduled independently and in a consecutive manner, one by one. Some trains however may have a higher scheduling priority than others. For example, cancelling a train assigned to a linked departure could cause cancellation of the linked trains, cancelling a joint train will produce a higher cost, and some trains are consuming far fewer resources than others, etc. Trains are therefore scheduled in the following order:

1. assigned trains:
   (a) joint trains: two (or more) joint trains are using the same resources without conflict,
   (b) trains that may be departing immediate: only arrival and departure gates are used,
   (c) trains assigned to linked departures: uncovering a linked departure can cause more uncovered departures,
   (d) trains that do not require any maintenance,
   (e) remaining assigned trains,
2. unassigned trains:
   (a) joint trains,
   (b) remaining unassigned trains.

Constraints relative to linked departures are respected, e.g. if train $t_2 \in \mathcal{T}$ is linked to departure $d \in \mathcal{D}$ and $t_1 \in \mathcal{T}$ is assigned to $d$, then $t_1$ must come before $t_2$ in the given ordering.

| | | CPU time (seconds) | | |
|---|---|---|---|---|
| Inst | Objective | assign | schedule | total |
| B1 | 752 916 | 2 | 6 | 8 |
| B2 | 689 251 | 2 | 6 | 8 |
| B3 | 640 895 | 9 | 6 | 15 |
| B4 | 1 555 790 | 4 | 27 | 31 |
| B5 | 1 926 970 | 16 | 33 | 49 |
| B6 | 1 363 350 | 4 | 22 | 26 |
| B7 | 230 101 | 1 | 2 | 3 |
| B8 | 265 607 | 1 | 2 | 3 |
| B9 | 1 703 030 | 13 | 27 | 40 |
| B10 | 182 755 | 0.5 | 1 | 1 |
| B11 | 1 351 390 | 1 | 5 | 6 |
| B12 | 707 955 | 1 | 1 | 2 |

Table 4.2: First Feasible results on B instances. Greedy and MIP are used for assignment.

### 4.4.4 Iterative Improvement Procedure

By applying the assignment and scheduling procedures described in the previous sections, feasible solutions to this problem are obtained in less than one minute for all benchmarks proposed in the ROADEF/EURO Challenge, as illustrated in Table 4.2. Please note that the running time for each instance is significantly less than the computation time allowed during the competition (i.e. 600 seconds).
This section will propose an iterative procedure for improving the schedule. This procedure operates as follows:

- (1) schedule more trains by allowing conflicts on the track groups,
- (2) resolve conflicts by means of a local search,
- repeat steps (1)-(2) until the stopping criteria are met.

The entire solution procedure is illustrated in Figure 4.12.

#### 4.4.4.1 Feasible to infeasible solution with more trains

The first step of this improvement procedure consists of adding more trains (and departures) to the feasible schedule by allowing conflicts. For each train added,

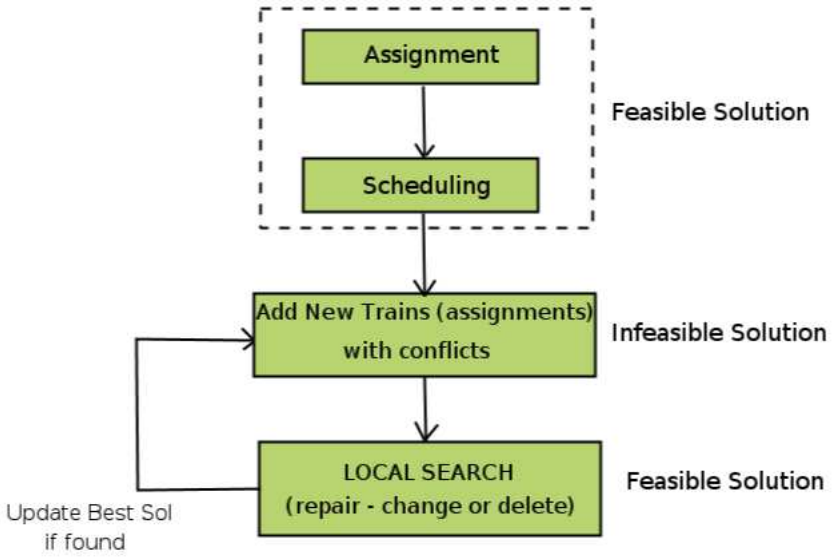


Figure 4.12: Solution Process

the allowed track group conflicts are limited to a given number (e.g. a maximum of 3 conflicts per train). This scheduling procedure is the same as the one previously explained, but without respecting the constraint on track group conflicts. All other constraints are to be respected. An infeasible solution generated in this manner will serve as input for the local search procedure described below.

#### 4.4.4.2 Local search to resolve track group conflicts

An infeasible solution is repaired by means of a local search procedure. The aim of this procedure is to change the choice of gates in order to reduce the number of conflicts to zero. The entry and exit times are to remain unchanged for each visit, as is the list of resources allocated for each train. Accordingly, a complete schedule for the train will either remain the same or be deleted (in the case of cancelling the train). The initial configuration (solution) is an infeasible set of visits on track groups. A visit is represented by a pair of gates $(g1, g2)$. Let's denote the configuration by $\mathcal{V} = \{(g1_1, g2_1), ..., (g1_n, g2_n)\}$ and the initial one by $\mathcal{V}^0 = \{(g1_1^0, g2_1^0), ..., (g1_n^0, g2_n^0)\}$. The domain of each variable $g1_i$ $(g2_i)$ includes all gates connecting the same pair of resources as $g1_i^0$ $(g2_i^0)$ and $NULL$ value. A visit corresponding to $(g1, NULL)$, $(NULL, g2)$ or $(NULL, NULL)$ is called a partial

visit. A configuration $\mathcal{V}$ is called *partial* if it contains a partial visit.

**Remark**: Two successive visits of the same train $(g1_i, g2_i)$ and $(g1_j, g2_j)$ share a common gate, i.e. $g2_i = g1_j$. For these cases, the local search procedure will perform the same modification at both gates.

The objective of the local search procedure is to minimize the number of cancelled trains. A train is cancelled if one of its visits is partial or should a related train be cancelled. Train $t_2$ is related to train $t_1$ if:

- $t_1$ and $t_2$ are joint (i.e. belong to the same joint arrival) or
- $t_1$ and $t_2$ correspond to the same physical unit (train) and $t_2$ arrives before $t_1$ (linked departures).

**Greedy procedure to resolve track group conflicts**

The first part of a local search is the greedy procedure to clear conflicts by deleting gates, i.e. setting the gate values to $NULL$. The objective here is to compute a partial, but feasible, configuration (set of visits). The heuristic is simple: delete the gate that will decrease conflicts by the greatest number until conflicts no longer exist.

**Tabu search on the partial feasible configuration**

The tabu search procedure starts from a partial, but feasible, configuration of gates given by the greedy-clear procedure. The goal is to assign gate values to partial visits while keeping the configuration feasible. This tabu search is very similar to the one presented in Chapter 2 for solving the Bin Packing Problem: similar moves, tabu tenure, choice of the move, etc.
The following two *elementary moves* are carried out:

- **ADD** gate corresponds to one of the 2 possible moves:
  - $(NULL, g2_i) \Rightarrow (g1_i, g2_i)$ ($g1_i \neq NULL$ and $g1_i$ leads to the same neighbor resource as $g1_i^0$)

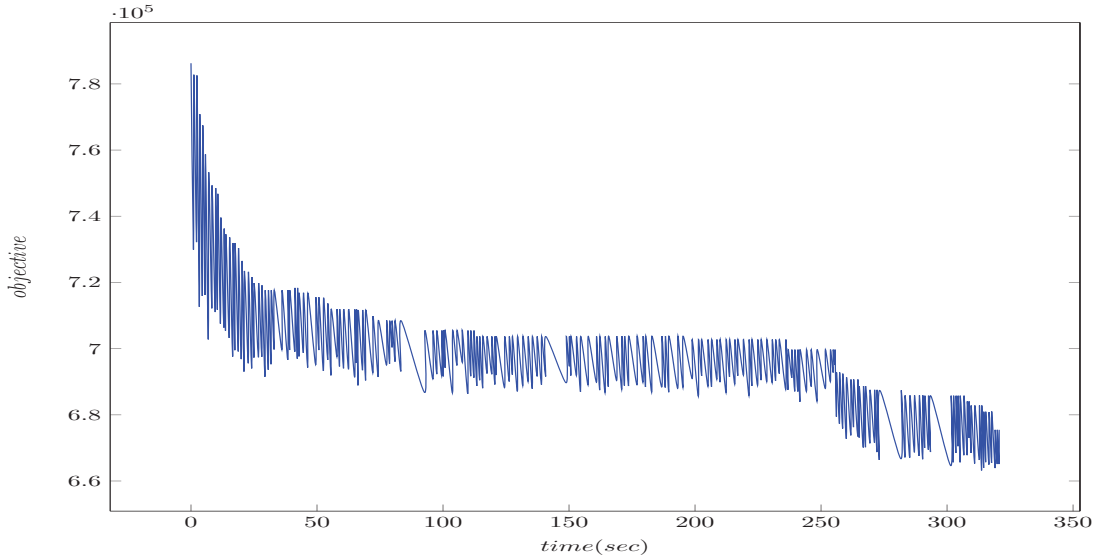


Figure 4.13: Objective function oscillation during improvement phase for instance B1. x-asis represents time elapsed since improvement procedure start, while y-axis represents the value of objective function after adding new trains and after local search procedure.

  - $(g1_i, NULL) \Rightarrow (g1_i, g2_i)$ ($g2_i \neq NULL$ and $g2_i$ leads to the same neighbor resource as $g2_i^0$)

- **DROP** gate corresponds to one of the 2 possible moves:

  - $(g1_i, g2_i) \Rightarrow (NULL, g2_i)$ ($delete\ g1$)
  - $(g1_i, g2_i) \Rightarrow (g1_i, NULL)$ ($delete\ g2$)

These modifications are also applied to the next/previous visit of the given train (see remark in Section 4.4.4.2).
The local search move consists of a single ADD move and, should conflicts occur, is to be followed by a few DROP moves in order to clear the conflicts. Deleting a gate $g_i$ (DROP) is allowed only if $g_i$ has not been added for a given number of iterations (i.e. if setting $g_i$ to $NULL$ is not tabu). The number of iterations for which deleting a gate is tabu equals to the frequency of adding this gate.

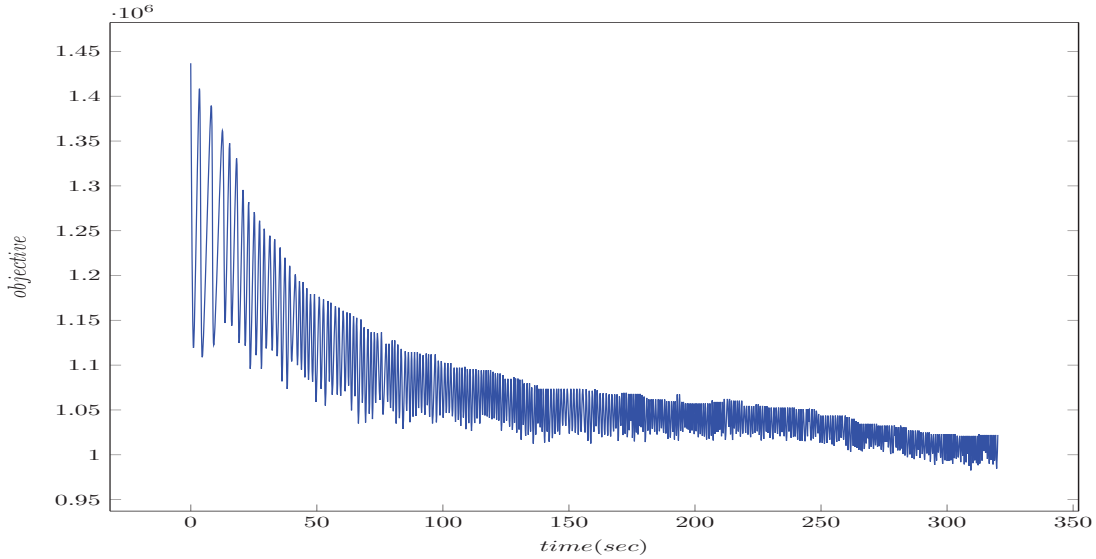


Figure 4.14: Objective function oscillation during improvement phase for instance X4.

The current configuration is evaluated by the following hierarchical function:

1. the number of trains cancelled,

2. the number of deleted gates, i.e. the number of gates with a NULL value.

At each iteration, a non-tabu move that minimizes this function is performed. Should two or more moves with the same objective exist, then a random choice is made. This process repeats until non-tabu move exists or until a maximum number of moves without improvement has been reached. In all reported experiments this limit has been set to 300.

The possibility of changing the arrival/departure platform for each train is also included in the local search. Constraints regarding platforms (length and conflict constraints) are respected during each of the moves; hence, checking platform constraints is part of the evaluation of the ADD move for the gate connected to the platform. Introducing the *platform change move* into the model has improved results, while slightly increasing the running time of the local search procedure.

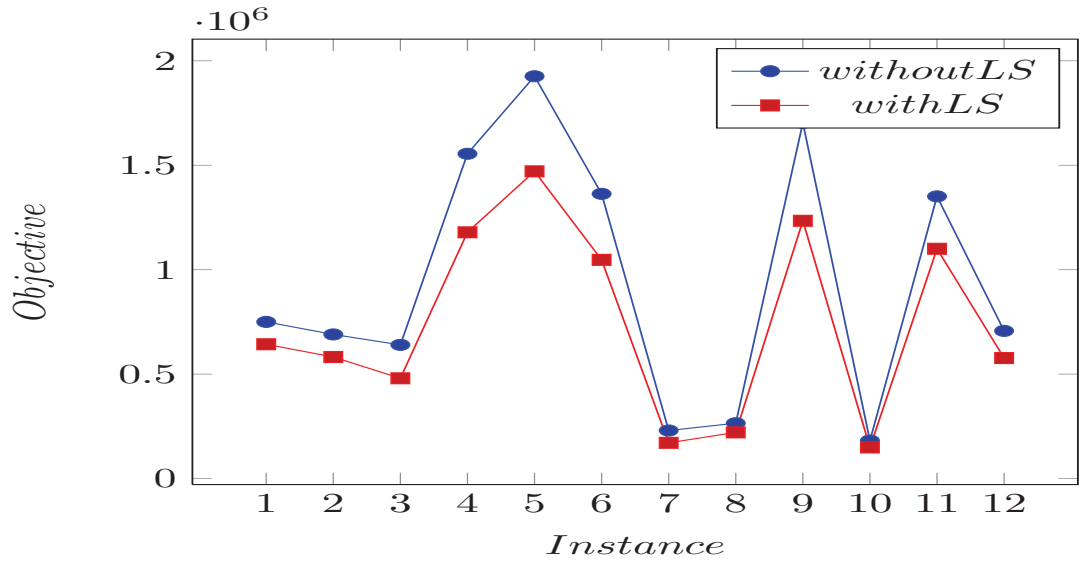


Figure 4.15: Improvement by local search (instances B1 – B12)

An illustration of the objective function change during this improvement procedure for two instances (`B1` and `X4`) is given in Figures 4.13 and 4.14. It can be noted that most of the improvement occurs during the early stage and not many new trains are added at the end. The improvement in results by the local search for each instance in dataset B is illustrated in Figure 4.15; the same improvement can be observed by comparing results in Tables 4.2 and 4.4 (column 5). Improvement varies between 14.2% and 27.0% over dataset B instances, with average improvement per instance being 20.8%.

**MIP instead of Local Search** The problem of repairing conflicts explained above and solved by local search procedure can possibly be solved in different ways. One possible approach that we have experimented is to define a problem as a mixed integer program (which is not too complicated), with the objective of determining the maximum subset (in terms of number of scheduled trains) of train schedules without conflicts on track groups. Whole improvement procedure remains the same, we only try to solve the problem of "repairing conflicts" by using mixed integer programming instead of local search.
If we consider a single iteration of improvement phase (add trains + repair) i.e. solving the problem – given a set of train schedules with conflicts, choose the gate

values such that minimum number of departures/arrivals is canceled – it is clear that MIP will produce better (or at least the same) solutions than local search. However, using MIP instead of local search has not made any (or negligible) improvements on the final results using presented improvement framework, especially in 10 minutes running time frame when using MIP usually produces worse results. The main reason for this is the fact that after many iterations of improvement phase (for ex. 100) solution values obtained by local search come very close to the values of infeasible solutions, i.e. only few trains can be added to the feasible solutions. Also, much more iterations can be done in the same running time if local search is used, which is crucial for some instances. Similar conclusions can drawn when MIP is used with a given running time limit (thus, transforming exact solution approach into heuristic one).

### 4.4.5 Final Algorithm

The pseudo-code of the final algorithm is given in 4.2.

**Algorithm 4.2:** *FinalAlgorithm*

```
1  Solve assignment problem (greedy/MIP);
2  Determine maintenance days;
3  Sort trains for schedule (see schedule order);
4  Add virtual visits;
5  for t ∈ 𝒯 do
6  |   schedule train t : schedule(t, d(t));
7  |   if train is not scheduled to departure cancel it;
8  repeat
9  |   add trains with conflicts;
10 |   repair conflicts (local search);
11 until stopping criteria met;
12 Park non-scheduled trains;
```

## 4.5 Evaluation and Computational results

### 4.5.1 Benchmarks

This method has been tested on the official set of competition instances provided by SNCF. This test data consists of two sets of instances, i.e. dataset B and dataset X. Both sets have been used to conduct the final evaluation. The first set was made available during the competition, while dataset X is a hidden set of instances provided to test algorithm robustness and then made publicly available after the end of the competition. Each set contains 12 instances and has been generated very similarly. For the sake of simplicity therefore, many experimental results in this chapter are given solely for dataset B, while very similar results are obtained on X. Four different resource infrastructures have been introduced, with six instances for each one - the first three instances from B and X (B1, B2, B3, X1, X2, X3) feature the same resource infrastructure, and the second three from each set have another infrastructure, etc. The basic characteristics of instances B are given in Table 4.3, while the instances from dataset X are very similar.

| Instances | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Inst | nbDays | \|A\| | \|D\| | \|TI\| | \|JA\| | \|JD\| | $maxMaint$ | \|L\| |
| B1 B2 | 7 | 1235 | 1235 | 37 | 281 | 237 | 30 | 475 |
| B3 | 7 | 1235 | 1235 | 37 | 281 | 237 | 60 | 0 |
| B4 B6 | 7 | 1780 | 1780 | 35 | 353 | 342 | 50 | 722 |
| B5 | 7 | 2153 | 1780 | 57 | 431 | 401 | 60 | 720 |
| B7 B8 | 1 | 304 | 304 | 33 | 40 | 28 | 100 | 143 |
| B9 | 7 | 1967 | 1967 | 52 | 296 | 239 | 100 | 860 |
| B10 | 1 | 196 | 196 | 31 | 56 | 58 | 20 | 90 |
| B11 | 7 | 1122 | 1122 | 43 | 314 | 317 | 20 | 486 |
| B12 | 3 | 570 | 570 | 40 | 160 | 167 | 20 | 263 |

Table 4.3: **Instances B:** #arrivals (|A|), #departures (|D|), #initial trains (|I|), #joint arrivals (|JA|), #joint departures (|JD|), maximum number of maintenances in a day ($maxMaint$), # linked departures (|L|).

### 4.5.2 Evaluation and results

This section will report on the computational experiments performed with the presented method on the given set of instances. The method has been implemented in C++ and compiled using Linux gcc 4.7.2 compiler in Ubuntu 12.10. All tests were performed on a computer with an Intel Core i7-3770 CPU 3.40 GHz processor. The mixed integer programming model proposed for the assignment phase has been solved using the IBM ILOG CPLEX 12.6 solver. The computation times reported here are given in seconds.
The algorithm sequentially produces independent solutions until the time limit is exceeded and retains the best one. Results obtained on datasets B and X (used for the final competition evaluation) are listed in Table 4.4. The results submitted to the competition are being reported, along with the improved results obtained following the competition deadline. The improvement was mainly achieved by using the MIP model, which was not included in the final submission, for the assignment problem. The results could be obtained in a running time of 10 minutes, thus satisfying the competition rules. These results can only be slightly improved with additional running time and were therefore not reported. A comparison is drawn with the best solutions obtained during the competition.
Since the number of uncovered arrivals and departures is the most important part of the objective, Table 4.5 reports the values corresponding to the best solutions obtained on dataset B. The values obtained by optimizing just the total number of uncovered arrivals/departures, i.e. in neglecting all other objectives, has also been reported. It can be noted that these values do not change for instances B7-B12, since this part represents a rather large weight in the objective function, corresponding to uncovering arrivals/departures, when compared to the other weights in the objective. Let's also note that the percentage of arrivals and departures cancelled is quite high, varying from 11.90% on the B3 instance to 44.21% on B12. The average percentage on dataset B is 26.04%. Train cancellation is mainly due to the fact that track group conflicts are prohibited. As mentioned in the problem statement, these conflicts are not always the real conflicts due to a "black-box" approach introduced to model track groups in the current problem. It is important to mention that conflicts have been modeled as a soft constraint (with the

| Instance | Challenge | | Improved results | |
|---|---|---|---|---|
| | Our Result | Best Result | Average | Best |
| B1 | 699 750 | 699 750 | 643 651 | 622 879 |
| B2 | 636 550 | 636 550 | 582 000 | 561 579 |
| B3 | 545 974 | 545 974 | 480 960 | 462 848 |
| B4 | 1 263 764 | 1 263 764 | 1 179 584 | 1 142 660 |
| B5 | 1 573 290 | 1 573 290 | 1 471 631 | 1 417 578 |
| B6 | 1 097 572 | 1 097 572 | 1 047 493 | 993 580 |
| B7 | 168 369 | 168 369 | 171 257 | 167 993 |
| B8 | 213 190 | 213 190 | 221 480 | 208 573 |
| B9 | 1 332 256 | 1 332 256 | 1 234 501 | 1 130 644 |
| B10 | 168 457 | 155 100 | 149 301 | 142 600 |
| B11 | 1 192 687 | 1 142 072 | 1 100 200 | 1 076 260 |
| B12 | 620 527 | 571 497 | 577 000 | 556 853 |
| X1 | 790 506 | 790 506 | 712 260 | 683 939 |
| X2 | 1 176 901 | 1 176 901 | 890 666 | 874 014 |
| X3 | 735 579 | 735 579 | 669 864 | 655 220 |
| X4 | 1 109 468 | 1 109 468 | 1 026 635 | 983 693 |
| X5 | 1 012 268 | 1 012 268 | 925 955 | 884 348 |
| X6 | 943 024 | 943 024 | 822 780 | 768 395 |
| X7 | 1,642,024 | 1 642 024 | 1 569 772 | 1 543 090 |
| X8 | 534 889 | 534 889 | 560 170 | 553 427 |
| X9 | 732 818 | 732 818 | 748 539 | 711 395 |
| X10 | 193 210 | 184 022 | 180 707 | 168 407 |
| X11 | 1,107 732 | 988 996 | 1 018 810 | 965 892 |
| X12 | 501 218 | 467 605 | 477 634 | 455 736 |

Table 4.4: **Results on datasets B, X** : average and best values from 20 runs of 10 minutes are reported for the improved results

corresponding cost in the objective) during the early stage of the competition, though this was later changed because of the large (and thus unrealistic) number of conflicts occurring in the solutions submitted. An early version of the method proposed herein, based on similar ideas and used to solve an earlier problem (with conflicts as a soft constraint), yielded results without any cancelled arrivals and less than 1% cancelled departures.

| Inst | Uncov. Dep. | | Uncov. Arr. | | Upper Bounds | |
|---|---|---|---|---|---|---|
| | # | % | # | % | #dep | #arr |
| B1 | 212 | 17.16 | 200 | 16.19 | 194 | 186 |
| B2 | 212 | 17.16 | 200 | 16.19 | 194 | 186 |
| B3 | 147 | 11.90 | 144 | 11.66 | 140 | 140 |
| B4 | 428 | 24.04 | 400 | 22.47 | 368 | 337 |
| B5 | 537 | 24.80 | 516 | 23.97 | 475 | 450 |
| B6 | 369 | 20.73 | 322 | 18.09 | 379 | 271 |
| B7 | 73 | 24.01 | 58 | 19.07 | 73 | 58 |
| B8 | 90 | 29.60 | 78 | 25.66 | 90 | 78 |
| B9 | 448 | 22.77 | 446 | 22.67 | 448 | 446 |
| B10 | 64 | 32.60 | 60 | 30.61 | 64 | 60 |
| B11 | 489 | 43.50 | 477 | 42.51 | 489 | 477 |
| B12 | 252 | 44.21 | 249 | 43.68 | 252 | 249 |

Table 4.5: **Uncovered arrivals/departures** : Number (and percentage) of uncovered arrivals and departures corresponding to the best solutions from table 4.4 are reported. Values obtained when considering number of uncovered arrivals/departures as the only objective are reported in the last two columns.

## 4.6 Qualification version

In this section we present the method for solving a qualification variant of the problem. Complete description of final version of the problem has been given in Section 4.2 and we will now list the differences occurring in the qualification version of the problem. They consist of turning some soft constraints into hard ones and vice versa and redefining the objective function:

- all arrivals and initial trains have to be scheduled i.e. must not have an empty schedule;
- conflicts on track groups and yard overcapacity are allowed;
- the following lexicographic objective is to be minimized:
  - number of cancelled departures
  - number of conflicts on track groups and yard overloads

– operational costs: $f^{plat}, f^{over}, f^{jun}, f^{pref}, f^{reuse}$

The same, two-phase approach has been developed for solving this variant of the problem, but with appropriate modifications to deal with different objective and constraints. As defined above, number of cancelled departures should be minimized first, followed by number of conflicts and yard overloads and operational costs. For the set of instances proposed by SNCF for the qualification stage of the challenge, the first objective part turned out to be the only relevant one. This is due to the difficulty of the proposed set of instances regarding the first objective. Moreover, all the teams have been ranked at the end of the qualification stage of the competition only according to this objective. Therefore, the only objective considered here is the number of cancelled departures. Contrary to departures, all arrivals have to be scheduled i.e. cancelling an arrival is not allowed. Gates choice on track group resources will be made randomly since conflicts are allowed and, as assumed, penalty in the objective function corresponding to those conflicts is neglected. The same rule applies for yard overloads. The method will be tested on benchmark dataset A, the one used in qualification stage of the competition.

### 4.6.1 Assignment problem

This section will describe the method adopted to solve the problem of matching (assigning) trains to departures. Obviously, junction or disjunction operation is required when assigning a single train to joint departure and vice-versa. For example, if two single trains $t_1$ and $t_2$ are assigned to joint departures $d_1, d_2 \in jointDep_d$, junction operation is required before the departure. If trains belonging to the same joint arrival are to be scheduled to departures belonging to the same joint departure, then no junction or disjunction operation is required. The need of performing junction of disjunction operations when assigning the train $t$, belonging to joint arrival $jointArr_a$, to departure $d$, belonging to joint departure $jointDep_d$, is not obvious, i.e. it depends of assignments of other trains belonging to the same joint arrival. Therefore, we will define feasible assignments for a train-departure pair $(t, d) \in \mathcal{T} \times \mathcal{D}$ and feasible assignments for a pair of joint arrival and joint departure $(jointArr_a, jointDepd_d)$.
We call an assignment $(t, d)$ of a train $t \in \mathcal{T}$ to a departure $d \in \mathcal{D}$ (called "single

assignment") *feasible* if the following holds:

1. $cat_t$ is compatible with $d$, i.e. $cat_t \in compCatDep_d$,
2. $arrTime_t + minDwell(t,d) + junDisjTime(t,d) + maintTime(t,d) + addTime(t,d) \leq depTime_d$,
3. $remDBM_t \geq reqD_d, remTBM_t \geq reqT_d$,

where:

- $minDwell(t,d)$, $maintTime(t,d)$ $addTime(t,d)$ are defined as before and
- $junDisjTime(t,d)$ is the total junction/disjunction duration required for scheduling $t$ to $d$: disjunction is required if $t$ is a part of joint arrival and junction is required if $d$ is a part of joint departure.

Assignment of joint arrival $jointArr_a$ to joint departure $jointDepd_d$ is called feasible if the following holds:

1. number of trains in $jointArr_a$ is equal to the number of departures in $jointDepd_d$, i.e. $|jointArr_a| = |jointDepd_d|$,
2. all corresponding single assignments are feasible (according to the previous definition) when $junDisjTime(t,d) = 0$.

The following lexicographic objective is considered during the assignment phase:

1. maximize the number of assigned departures,
2. minimize the number of maintenance operations.

Second objective is introduced in order to minimize the use of facility resources, which showed to be critical resources.
As before, greedy and MIP procedures have been used to solve the assignment problem. Additionally, combining these two procedures with minimum weighted matching algorithms has also been implemented.

#### 4.6.1.1 Greedy assignment algorithm

Algorithm is given in 4.3. When looking for the best train for each departure $d \in jointDep_d$, the attempt is first made to find a single feasible assignment $(t, d)$ and, in case the one is not found, we look for a feasible joint assignment $(jointArr_a, jointDep_d)$.

When looking for the best train for a linked departure $d$, one can choose the train with minimum value $v$ ("smallest" train) and perform a maintenance operation or choose the "biggest" train (train with maximum value $v$) without performing a maintenance. In our experiments, the first choice gives better results than the second one. Nevertheless, one can decide to use both of the two options during the procedure. Thus, we randomly pick one of the two choices in each iteration (for each departure $d$) with a probability of choosing the first one being significantly greater than the second one (0.8 and 0.2 for example). A great variation can occur in obtained results for some of the instances: for example, number of non-assigned departures for benchmark A1 varies from 33 to 50. Greedy procedure is run several times with different random seeds and the best assignment is chosen.

For certain benchmarks i.e. those for which daily maintenance limit constraint is tight, the greedy procedure is combined with matching in order to improve the quality of assignments, as will be explained in the following section.

#### 4.6.1.2 MIP formulation for assignment

MIP model analogous to the one used for the final variant of the problem is developed. The only differences are non-existence of constraints for forbidding junction and disjunction operations and the objective function. The objective function here is :

$$\text{Maximize} \quad \sum_{j=1}^{n} |T_j| \times x_j, \quad x_j \in \{0, 1\} \qquad j = 1, ..., n$$

**Algorithm 4.3:** Greedy assignment

```
1  Sort departures by time;
2  for d = 0 to |𝒟| − 1 do
       // each departure
3      bestTrain ← −1, minValue ← 10000000;
4      for t = 0 to |ℱ| − 1 do
5          isFeasible ← (d(t) = −1) ∧ isFeasible(t, d) ∧ checkMaintenanceLimit();
6          if isFeasible then
7              value ← 10000000;
8              nmbM ← calculateNmbRequiredMaintenances(t, d);
9              value = min(remDBM_t / reqD_d, remTBM_t / reqT_d);
10             if d is not linked and nmbM > 0 then
11                 value ← value + nmbM ∗ 100000;
12             if d is linked and nmbM > 0 then
13                 value ← value − 100000;
14             if d is linked and nmbM = 0 then
15                 value ← −value;
16         if value < minValue then
17             minValue ← value;
18             bestTrain ← t;
19     if bestTrain ≠ −1 then
20         t(d) ← bestTrain, d(bestTrain) ← d;
21         Update linked arrival if needed;
22         Add maint. to all intervals containing [arrDay, depDay] if required;
```

#### 4.6.1.3 Greedy assignment + matching + MIP

To obtain improved solutions to the assignment problem, the greedy procedure explained above is combined with a weighted matching algorithm in a following way:

1. the assignment problem is solved by a greedy procedure;
2. assignments of linked departures are fixed (in updating the data on linked trains);
3. solve remaining assignment problem by matching, minimizing a given lexicographic objective, with the following restriction: train $t$ can have a maintenance in new assignment $(t, d')$ only iff train $t$ also had a maintenance in previous (greedy) assignment $(t, d)$;
   - restriction assures that maintenance limit constraint remains satisfied
   - number of maintenances has possibly decreased
4. add more assignments (as in greedy) if possible;
5. repeat previous steps until no new assignments can be added.

Restriction has been made in step 2 since matching algorithm cannot deal with maintenance limit constraint. New assignments can possibly be added in step 3 of the procedure since number of maintenances has possibly decreased by using the matching. One can note that this improvement procedure will not be useful if daily maintenance limit has not been reached in the solutions obtained by greedy procedure.

Assignment values obtained by described method on dataset A are listed in Table 4.6. One can note that greedy assignment can be improved only on instances A1, A2, A4-A6. This is due to the fact that daily maintenance limit constraint is not tight (can almost be ignored) for remaining instances, i.e. maximum number of maintenances in a day is large enough and is never reached. Linked departures do not exist in A3 and A9. Additional time ($addTime$) for assignments used in

| Assignment results | | | | |
|---|---|---|---|---|
| Inst | G | G_MIP | G_M_MIP | LB |
| A1-2 | 36 | 25 | 20 | 6 |
| A3 | 12 | 12 | 12 | 12 |
| A4 | 38 | 27 | 20 | 6 |
| A5 | 28 | 17 | 11 | 5 |
| A6 | 39 | 27 | 20 | 6 |
| A7-8 | 4 | 4 | 4 | 0 |
| A9 | 11 | 11 | 11 | 0 |
| A10 | 4 | 4 | 4 | 0 |
| A11 | 3 | 3 | 3 | 0 |
| A12 | 4 | 4 | 4 | 0 |

Table 4.6: **Assignment values**. Number of non-assigned departures is given for each instance and each method used: G - greedy,G_MIP - greedy + MIP, G_M_MIP - greedy + matching + MIP. $addTime = 5minutes$ is used

experiments is 5 minutes.
A simple lower bound on number of non-assigned departures reported in Table 4.6 is calculated in the following way:

- Use maximum possible $remDBM$ and $remTBM$ for each train $t$;
  - If $t$ is not linked, use original $remDBM$ and $remTBM$
  - If $t$ is linked with departure $d$ then:
    $remDBM_t = maxDBM_t - reqD_d$
    $remTBM_t = maxTBM_t - reqT_d$
- Build a bipartite graph as described before without junction(disjunction) time and additional time (i.e. $addTime = junDisjTime = 0$) and using calculated $remDBM$ and $remTBM$ for each train;
- Find maximal matching in a graph (maintenance constraint not included).

### 4.6.2 Scheduling problem

Contrary to the final version of the problem, two possibilities exist for the schedule of each train $t \in \mathcal{T}$:

1. $t$ is scheduled to departure $d \in \mathcal{D}$;

2. $t$ is parked inside the station until the end of the planning time frame without being assigned to any departure.

All junction and disjunction operations are performed on platforms, just after the arrival or just before the departure. **Move from one facility to another** has been added to the list of possible train movements.

#### 4.6.2.1 Choosing platforms, parking resources and travel starting times

As before, ideal starting time, $idealST$, for each travel is determined, and procedure will try to find a feasible travel with starting time as close as possible to $idealST$. In this variant of the problem, ideal starting time chosen for each travel is the earliest or latest possible time, depending on the type of travel. The reason for this is to minimize the usage of critical resources (especially platforms and facilities). For example, the train arriving to the platform will leave the platform as soon as possible and train to depart from a platform will arrive to the platform as late as possible. An important constraint of the problem is that all arrivals have to be scheduled. This means that available platform has to be found for each arrival. Similarly, parking resource (if necessary) has to be found, but this is less critical since yard capacity is not a hard constraint. Therefore, the following strategy is used for arriving trains:

- arrivals are sorted in descending order by train length and scheduled one by one in this order (sorting should respect "linked" constraints),

- available platforms are sorted in ascending order by length for each arrival/departure and the first one is chosen.

Similarly, maintenance facility with minimum length is chosen for each maintenance operation.

When movement to the yard i.e. parking has to be made, the yards are sorted randomly and all paths are checked until a feasible travel has been found.
Scheduling procedure is run several times with different random seeds, with slightly different parameters such as random ordering of yards, slick time choice, choice of platform /facility if there are several "equal" choices, etc., and the best one is retained.

### 4.6.3 Evaluation and Computational results

| Instances A | | | | |
|---|---|---|---|---|
| inst | $\|\mathcal{T}\|$ | $\|\mathcal{D}\|$ | $maxMaint$ | L |
| A1 A2 A4 A6 | 1272 | 1235 | 30* | 35% |
| A3 | 1272 | 1235 | 60 | 0% |
| A5 | 1534 | 1499 | 40* | 30% |
| A7 A8 A10 A12 | 1815 | 1780 | 50 | 40% |
| A9 | 1815 | 1780 | 100 | 0% |
| A11 | 2210 | 2153 | 60 | 33% |

Table 4.7: Instances A1 – A12: $|\mathcal{T}|$ - number of trains (initial + arrivals), $|\mathcal{D}|$ - number of departures, $maxMaint$ - maximum number of maintenances in a day (* means that constraint is tight), L - percentage of departures that are linked.

This method has been tested on the official set of twelve competition instances (A1 – A12) provided by SNCF for a qualification phase. Algorithm sequentially produces independent solutions until the time limit is exceeded and keeps the best one. Assignment problem is solved several times and the best solution is taken. For a given assignment several schedules are made and the best one is taken. Computing time for one feasible solution (Assignment + Schedule) is few seconds. Computational results on instances set A are given in the table 4.8. Comparison is made with the best solutions obtained in qualifying stage of the competition (Q).

| Solutions A - 10 minutes, 10 runs | | | | | |
|---|---|---|---|---|---|
| Inst | Average | Best | Assignment | Q | LB |
| A1 | 23.2 | 20 | 20 | 46 | 6 |
| A2 | 23.2 | 20 | 20 | 46 | 6 |
| A3 | 12.1 | 12 | 12 | 15 | 12 |
| A4 | 25.4 | 22 | 20 | 31 | 6 |
| A5 | 12.5 | 11 | 11 | 25 | 5 |
| A6 | 25.1 | 21 | 20 | 32 | 6 |
| A7 | 4.1 | 4 | 4 | 8 | 0 |
| A8 | 4.1 | 4 | 4 | 8 | 0 |
| A9 | 11.7 | 11 | 11 | 15 | 5 |
| A10 | 6.6 | 5 | 4 | 12 | 0 |
| A11 | 4.0 | 4 | 3 | 15 | 0 |
| A12 | 6.5 | 6 | 4 | 12 | 0 |
| Total | 158.5 | 140 | | 265 | |

Table 4.8: Results A: average and best result over 20 runs of 10 minutes is reported in second and third column. For each solution, 50 assignments with different random seeds and 3 schedules for the best assignment are generated. Assignment value (result of assignment procedure) in the best solutions is given in the fourth column. The best results for each instance obtained in ROADEF/EURO Challenge are reported in the last column.

## 4.7 Conclusion

This rolling stock unit management problem on a railway site is extremely difficult to solve for several reasons. Most induced sub-problems, such as the assignment problem, scheduling problem, track group conflict problem and platform assignment problem, are indeed complicated. In order to solve this problem, we have proposed a two-phase approach that combines exact and heuristic methods.

A natural way of approaching the problem consists of dividing it into two sub-problems, the first consisting of matching (assigning) trains to departures and the second consisting of planning train movements (scheduling) inside the station, and then solving both sequentially. The presence of linked departures and a constraint

on the daily maintenance limit further complicate the assignment problem. Otherwise, the problem could be solved in a polynomial time by means, for instance, of the maximum weighted matching algorithm. A common strategy for overcoming these constraints calls for modeling the assignment problem as a mixed integer program. The number of linked departures in the given set of instances however makes the MIP model unworkable due to the tremendous number of variables needed to be generated in order to cope with the linked departures. A greedy procedure has therefore been used to solve the assignment problem. The departures are sorted by departure time and matched to the trains one by one, in a greedy manner. A simple function has been introduced to evaluate the quality of assignment $(t, d)$. Solutions to the assignment problem are improved by combining a greedy procedure with MIP, whereby all train assignments with linked departures obtained by the greedy procedure are fixed and the remaining assignment is solved by MIP. We have proposed the simple idea (and proven its correctness) of modeling the constraint on daily maintenance limit as a linear one.

The second step in solving the problem is to plan train movements inside the station in respecting all resource constraints and cancelling as few trains as possible. This problem is very complex and features a tremendous number of decision variables (all resources, gates and entry/exit times must be specified for each train); hence, modeling the problem exactly and solving it efficiently are likely to be impossible. We have therefore opted for a constructive procedure to schedule the trains. They are scheduled sequentially, one by one, after being ordered by a given set of criteria. The possible train movements are restricted to just a few of critical importance in order to reduce problem complexity. For this purpose, a set of potential paths trains are allowed to use has been constructed at the beginning of the procedure.
A concept of virtual visits has been offered to expand the choice of starting times for each travel in order to address track group conflicts. An iterative procedure has been adopted to improve the scheduling phase solutions, by allowing infeasible solutions as regards track group conflicts and then resolving these candidate solutions by means of a local search. The introduction of both virtual visits and an iterative improvement procedure has served to significantly improve these so-

lutions.

The algorithm described in this chapter was ranked first at the ROADEF/EURO Challenge 2014 competition, recognized as the best solution for 18 out of the 24 competition instances. A number of simplifications have been performed for the final submission in order to enhance method reliability. In deleting these simplifications, the method can indeed be improved. Allowing violation of some other constraints such as resource length and capacity (probably a "small" violation) could possibly be included in proposed improvement heuristic and might improve the quality of final results. We decided to allow only violations of conflict constraint since this constraint showed to be, by far, the most restrictive for proposed set of instances.

# Conclusions

In this chapter we highlight the main contributions of our research and summarize the main results. As we stated earlier, the scope and the aims of this thesis are to investigate how to efficiently solve several difficult combinatorial optimization problems, by local search based heuristics. In addition to local search, greedy algorithms and Mixed Integer Programming (MIP) have been used in order to produce initial feasible solutions for one of the considered problems. Performance of local search approaches is influenced by several important factors such as a set of neighborhoods to be explored, intensification and diversification strategies, implementation efficiency, etc. Simple (or medium size) neighborhoods for each of the considered problems have been used in proposed approaches, with high solutions quality being obtained by adding several local search features such as intensification and diversification strategies, noising procedures, restarts, tabu list, etc.

Two of considered problems are a very large scale problems arising from real-world applications where essential and complex features of problems are present: Machine Reassignment problem defined by Google and Rolling Stock Problem defined by French Railways (SNCF). The third problem considered here is One-dimensional Bin Packing Problem (BPP), a classical and well known combinatorial optimization problem. Solving these problems is computationally challenging. For the first two problems this is mainly due to the fact that problems are large scale and contain many constraints, while for BPP main difficulty comes from the fact that competitive method is required to find optimal solutions for most of the instances.
To tackle these problems efficiently, in this thesis we constructed solution methods based on local search. Obtained results are competitive with the best results found

in the literature (or proposed by competitors in ROADEF/EURO Challenge competitions): most of the obtained results are proven to be optimal, near optimal, or the best known.

In Chapter 2, local search approach to One-dimensional Bin Packing problem (BPP) based on exploring partial solutions is proposed to solve the problem successfully. As opposed to the majority of work published on BPP, a local search explores partial solutions that consist of a set of assigned items without any capacity violation and a set of non-assigned items. The main contribution is a Tabu search on partial and consistent solution (called Consistent Neighborhood Search) that includes moves consisting of rearranging both the items assigned to a single bin and non-assigned items, i.e. adding and dropping items to and from the bin (*Swap* move). *Swap* move includes maximum three items from a single bin. This higher complexity of *Swap* moves, with respect to the classical (mostly shift and exchange) moves used in the literature, is compensated by the fact that only moves between assigned and non-assigned items are performed.
Some of the algorithm features crucial for obtaining high quality solutions include:

- size of a partial solution and termination criteria i.e. exploring partial solutions with $m-2$ bins and terminating the search when all non-assigned items can be packed into two bins (thus, producing complete feasible solution with $m$ bins),
- defining a suitable fitness function i.e lexicographic fitness function minimizing total weight of non-assigned items first and maximizing a number of non-assigned items second
- introducing second tabu search variant consisting of a subset of moves that do not decrease the second objective,
- allowing only the moves between assigned and non-assigned items speeds up the search significantly.

Promising results have been obtained for a very wide range of benchmark instances; best known or improved solutions obtained by heuristic methods have been found

for all considered instances for BPP, successfully outperforming published results for the particular class of instances `hard28`, which appears to provide the greatest degree of difficulty for BPP algorithms. This method is also tested on vector packing problems (VPP) and evaluated on classical benchmarks for two-dimensional Vector Packing Problem (2-DVPP), in all instances yielding optimal or best-known solutions.

Local Search approach for Machine Reassignment Problem (MRP) has been proposed in Chapter 3. MRP shares some similarities with bin packing problems considered in Chapter 2, but contains additional constraints and has different objective function. Due to the size of the instances, exploring the search space "smartly" and efficiently was crucial in obtaining high quality results. Besides two simple neigborhoods, $shift$ and *swap*, used in most of the papers on MRP, a neighborhood referred as BPR (for Big Process Rearrangement) has been defined and showed to be very useful in obtaining better results. Limiting the neighborhood exploration (by maximum number of evaluations for example) has been done in order to deal with the size of instances. Several strategies have been developed in order to improve the results, including strategies for search intensification and diversification, defining a suitable order of processes when exploring the neighborhoods and restart procedures. Experiments on large-scale problem instances, proposed by Google and containing up to 50,000 processes and 5,000 machines, have been conducted, showing a remarkable performance of the presented local search algorithm. Proposed algorithm was ranked first in ROADEF/EURO Challenge 2012. It is important to mention that 50 teams have submitted their algorithms in the competition, with many proposed approaches being based on local search. Second and third placed team have proposed Large Neighborhood Search methods, using Constraint or Integer Programming to solve sub-problems.

Chapter 4 considers SNCF Rolling Stock management problem: rolling stock unit management on railway sites, defined by French Railways (SNCF). The problem is to manage trains between their arrivals and departures in terminal stations. The problem is very complex and involves temporary parking and shuntings on infrastructure which are typically platforms in stations, maintenance facilities,

railyards located close to train stations and tracks linking them. Proposed approach is a two-phase approach that solves two sub-problems sequentially. Several techniques are applied, including greedy algorithms, Mixed integer Programming (MIP) and Local Search. Feasible solutions to the problem have been obtained by greedy algorithms and MIP (only to obtain better initial solutions, not required). These solutions are then a subject of a local search based improvement procedure. More precisely, improvement procedure consists of iteratively producing infeasible solution (in terms of the most difficult, conflict, constraint) by adding more arrivals/departures to the schedule and repairing it by local search. This local search is Consistent Neighborhood Search, based on exploration of partial solutions (as in proposed methods devised for Set Covering and Bin Packing in Chapters 1 and 2). The method submitted to ROADEF/EURO 2014 competition was ranked first, obtaining the best results for 18 out of 24 instances, while improved version that includes MIP produces the best results on remaining six instances as well. To the best of our knowledge, no local search as an important method ingredient has been proposed by other competitors.

As a final remark, we would like to point out the importance of appropriate algorithm choices when designing the methods for solving proposed problems given that short running time limit is imposed (either by competition rules or in order to be competitive with the best approaches in the literature). Therefore, several important steps have been performed when designing the algorithms such as:

- carefully study the problem (instances, size of the problem, find greatest difficulties, etc.);
- try to find characteristics shared by all (or majority of) available instances of the problem and take them into account when design the algorithm;
- using appropriate data structures and fast algorithm implementation;
- defining the suitable termination conditions for each part of the algorithm;
- performing reductions or simplifications of the problem if possible and necessary;

- keeping only algorithm parts that contribute the most to quality of the final results i.e. dropping less important features in order to speed up the algorithm (based on extensive numerical experiments), ...

Obviously, some of these steps are to be done even when no time limit on running the algorithm exists, but existence of short time limit makes them particularly important. Many experiments have been performed for all presented problems (particularly for Bin Packing and Machine Reassignment), including different choices of neighborhoods and the way of their exploration, termination conditions, applying possible reductions and/or decompositions, etc. and the final algorithms, that showed to be the best in terms of solution speed and quality, have been presented in this thesis.

# References


Google roadef/euro challenge 2011-2012: Machine reassignment. 2012. URL http://challenge.roadef.org/2012/files/problemdefinitionv1.pdf. 56, 57

E.H.L. Aarts and J.K. Lenstra. *Local Search in Combinatorial Optimization*. John Wiley & Sons, 1997. 1

A. Alvim, C. Ribeiro, F. Glover, and D. Aloise. A hybrid improvement heuristic for the one-dimensional bin packing problem. *Journal of Heuristics*, 10:205–229, 2004. 28, 30, 35

Zahra Naji Azimi, Paolo Toth, and Laura Galli. An electromagnetism meta-heuristic for the unicost set covering problem. *European Journal of Operational Research*, 205(2):290–300, 2010. 17

G. Belov and G. Scheithauer. A branch-and-cut-and-price algorithm for one-dimensional stock cutting and two-dimensional two-stage cutting. *European Journal of Operational Research*, 171(1):85–106, 2006. 43

S. Binato, W. J. Hery, D. M. Loewenstern, and M. G. C. Resende. A greedy randomized adaptive search procedure for job shop scheduling. *IEEE Trans. on Power Systems*, 16:247–253, 2001. 8

Filipe Brandão and J. P. Pedroso. Bin Packing and Related Problems: General Arc-flow Formulation with Graph Compression. Technical Report DCC-2013-08, Faculdade de Ciências da Universidade do Porto, Portugal, 2013. 30, 44, 50

Felix Brandt, Jochen Speck, and Markus Volker. Constraint-based large neighborhood search for machine reassignment. *Annals of Operations Research*, pages 1–29, 2014. 63, 65

H. Cambazard and N. Catusse. Roadef challenge 2014: A modeling approach, rolling stock unit management on railway sites. *IFORS 2014, Barcelona 13 – 18 July*, 2014. 116

A. Caprara and P. Toth. Lower bounds and algorithms for the 2-dimensional vector packing problem. *Discrete Applied Mathematics*, 111(3):231–262, 2001. 31, 49

A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations Research*, 47:730–743, 1999. 17

Alberto Caprara, Matteo Fischetti, and Paolo Toth. Algorithms for the set covering problem. *Annals of Operations Research*, 98:2000, 1998. 17

I. Charon and O. Hurdy. The noising method. *EJOR*, 135:86–101, 2012. 79

Frank Dammeyer and Stefan Voß. Dynamic tabu list management using the reverse elimination method. *Annals OR*, 41(2):29–46, 1993. 19

J.M. Valerio de Carvalho. Exact solution of bin-packing problems using column generation and branch-and-bound. *Annals of Operations Research*, 86:629–659, 1999. 29

J.A. Diaz and E. Fernandez. A tabu search heuristic for the generalized assignment problem. *Technical Report DR.*, 98/8, 1998. 64, 69

ESICUP. Euro especial interest group on cutting and packing. one dimensional cutting and packing data sets. `http://paginas.fe.up.pt/~esicup/tiki-list_file_gallery.php?galleryId=1`, 2013. 43

E. Falkenauer. A hybrid grouping genetic algorithm for bin packing. *Journal of Heuristics*, 2:5–30, 1996. 43

S.P. Fekete and J. Schepers. New classes of fast lower bounds for bin packing problems. *Mathematical Programming*, 91:11–31, 2001. 28

T.A. Feo and M.G.C. Resende. Greedy randomized adaptive search procedures. *Journal of Global Optimization*, 6:109–134, 1995. 7, 8, 12, 14, 15, 17

P. Festa and M.G.C. Resende. Grasp: An annotated bibliography. *in Essays and Surveys on Metaheuristics, C.C. Ribeiro and P. Hansen (Eds.), Kluwer Academic Publishers*, pages 325–367, 2002. 8

K. Fleszar and C. Charalambous. Average-weight-controlled bin-oriented heuristics for the one-dimensional bin-packing problem. *European Journal of Operational Research*, 210:176–184, 2011. 30, 35, 43

K. Fleszar and K. Hindi. New heuristics for one-dimensional bin-packing. *Computers & Operations Research*, 29:821–839, 2002. 29, 30, 35

R. Freling, RM. Lentink, LG. Kroon, and D. Huisman. Shunting of passenger train units in a railway station. *Transportation Science*, 39(2):261–272, 2005. 116

M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN 0-7167-1044-7. 11, 25

F. Glover and M. Laguna. Tabu search. *Kluwer Academic Publishers*, 7:239–240, 1997. 19

J. Gupta and J. Ho. A new heuristic algorithm for the one-dimensional bin packing problem. *Production Planning & Control*, 10:598–603, 1999. 29, 35

J. Haahr and S.H. Bull. Exact methods for solving the train departure matching problem. *IFORS 2014, Barcelona 13 – 18 July*, 2014. 116

Djamal Habet and Michel Vasquez. Solving the selecting and scheduling satellite photographs problem with a consistent neighborhood heuristic. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004), 15-17 November 2004, Boca Raton, FL, USA*, pages 302–309, 2004. 3, 7, 28

Hideki Hashimoto, Sylvain Boussier, Michel Vasquez, and Christophe Wilbaut. A grasp-based approach for technicians and interventions scheduling for telecommunications. *Annals OR*, 183(1):143–161, 2011. 8

W. Jaskowski, M. Szubert, and P. Gawron. A hybrid mip-based large neighborhood search heuristic for solving the machine reassignment problem. *Annals of Operations Research*, pages 1–30, 2015. 63, 65, 69

J.E.Beasley. Or-library: distributing test problems by electronic mail. *J Oper Res Soc*, 41(11):1069–1072, 1990. 13, 14

David S. Johnson. *Near-optimal bin-packing algorithms*. Doctoral Thesis. MIT Press, Cambridge, 1973. 27

RM. Lentink, PJ. Fioole, LG. Kroon, and C. van't Woudt. Applying operations research techniques to planning of train shunting. *Technical Report ERS-2003-094-LIS, Erasmus University Rotterdam, Rotterdam, The Netherlands*, 2003. 116

K.H. Loh, B. Golden, and E. Wasil. Solving the one-dimensional bin packing problem with a weight annealing heuristic. *Computers and Operations Research*, 35(7):2283–2291, 2008. 30

Yannis Marinakis, Athanasios Migdalas, and Panos M. Pardalos. Expanding neighborhood grasp for the traveling salesman problem. *Comp. Opt. and Appl.*, 32 (3):231–257, 2005. 8

S. Martello and P. Toth. *Knapsack problems: algorithms and computer implementations*. Wiley, Chichester, England, 1990. 28, 29

Renaud Masson, Thibaut Vidal, Julien Michallet, Puca Huachi Vaz Penna, Vinicius Petrucci, Anand Subramanian, and Hugues Dubedout. An iterated local search heuristic for multi-capacity bin packing and machine reassignment problems. *Expert Syst. Appl.*, 40(13):5266–5275, 2013. 31, 50, 51, 64, 66, 69, 88

Deepak Mehta, Barry O'Sullivan, and Helmut Simonis. Comparing solution methods for the machine reassignment problem. In Michela Milano, editor, *Principles*

*and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 782–797. Springer Berlin Heidelberg, 2012. ISBN 978-3-642-33557-0. doi: 10.1007/978-3-642-33558-7_56. 62, 63, 65

M. Monaci and P. Toth. A set-covering-based heuristic approach for bin-packing problems. *INFORMS Journal on Computing*, 18(1):71–85, 2006. 31, 49

Bernhard Nebel, editor. *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence, IJCAI 2001, Seattle, Washington, USA, August 4-10, 2001*, 2001. Morgan Kaufmann. ISBN 1-55860-777-3. 19

James Ostrowski, Jeff Linderoth, Fabrizio Rossi, and Stefano Smriglio. Solving large steiner triple covering problems. *Oper. Res. Lett.*, 39(2):127–131, 2011. 17

Leonidas S. Pitsoulis, Panos M. Pardalos, and Donald W. Hearn. Approximate solutions to the turbine balancing problem. *European Journal of Operational Research*, 130(1):147–155, 2001. 8

Gabriel M. Portal, Marcus Ritt, Leonardo M. Borba, and Luciana S. Buriol. Simulated annealing for the machine reassignment problem. *Annals of Operations Research*, pages 1–22, 2015. 63, 69

William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C, 2nd Edition*. Cambridge University Press, 1992. 16

Marcela Quiroz-Castellanos, Laura Cruz Reyes, José Torres-Jiménez, Claudia Gómez Santillán, Héctor Joaquin Fraire Huacuja, and Adriana C. F. Alvim. A grouping genetic algorithm with controlled gene transmission for the bin packing problem. *Computers & OR*, 55:52–64, 2015. 30, 44

F. Ramond and N. Marcos. Roadef/euro 2014 challenge, trains don't vanish! - final phase, rolling stock unit management on railway sites. *SNCF - Innovation & Research Department*, 2014. URL `http://challenge.roadef.org/2014/files/Challenge_sujet_140224.pdf`. 92, 111

Armin Scholl, Robert Klein, and Christian Jurgens. Bison: A fast hybrid procedure for exactly solving the one-dimensional bin packing problem. *Computers & Operations Research*, 24(7):627 – 645, 1997. 29, 43

P. Schwerin and G. Wäscher. A new lower bound for the bin-packing problem and its integration into mtp and bison. *Pesquisa Operacional*, 19:111–129, 1999. 29, 43

A. Singh and A.K. Gupta. Two heuristics for the one-dimensional bin-packing problem. *OR Spectrum*, 29(4):765–781, 2007. 30

F. Spieksma. A branch-and-bound algorithm for the two-dimensional vector packing problem. *Computers & Operations Research*, 21(1):19–25, 1994. 31, 49

Michel Vasquez, Audrey Dupont, and Djamal Habet. Consistency checking within local search applied to the frequency assignment with polarization problem. *RAIRO - Operations Research*, 37(4):311–323, 2003. 3, 7, 28

Gerhard Wäscher and Thomas Gau. Heuristics for the integer one-dimensional cutting stock problem: A computational study. *Operations-Research-Spektrum*, 18(3):131–144, 1996. 43

M. Yagiura, T. Yamaguchi, and T. Ibaraki. A variable depth search algorithm with branching search for the generalized assignment problem. *Optimization Methods and Software*, 10:419–441, 1998. 64, 69

M. Yagiura, T. Ibaraki, and F. Glover. An ejection chain approach for the generalized assignment problem. *INFORMS Journal on Computing*, 16:133–151, 2004. 64, 69

M. Yagiura, M. Kishida, and T. Ibaraki. A 3-flip neighborhood local search for the set covering problem. *European Journal of Operational Research*, 172(2):472 – 499, 2006. 17

Snežana Mitrovic-Minic and Abraham P. Punnen. Local search intensified: Very large-scale variable neighborhood search for the multi-resource generalized assignment problem. *Discrete Optimization*, 6(4):370 – 377, 2009. 64

# Publications

### International journals

Mirsad Buljubasic and Haris Gavranovic, An efficient local search with noising strategy for Google Machine Reassignment problem. *Annals of Operations Research*, 2014 (published online).

### International journals - submitted or in revision

Mirsad Buljubasic and Michel Vasquez, Consistent Neighborhood Search for One-Dimensional Bin Packing and Two-Dimensional Vector Packing. *Computers & Operations Research.*

Mirsad Buljubasic and Michel Vasquez and Haris Gavranovic, Two Phase Heuristic for SNCF Rolling Stock Problem. *Journal of Heuristics.*

### Conferences

Two Phase Approach Combining Heuristic and Integer Programming for SNCF Rolling Stock Problem, *IFORS 2014, Barcelone 13-18 July*, 2014.

Two Phase Heuristic for SNCF Rolling Stock Problem, ROADEF 2015, Marseille 25-27 February, 2015.

### Book chapters

Michel Vasquez and Mirsad Buljubasic, Une procedure de recherche iterative en deux phases : la methode GRASP, *Metaheuristiques pour l'optimisation difficile, Eyrolles*, 2014.

**Abstract** This thesis focuses on the design and implementation of local search based algorithms for discrete optimization. Specifically, in this research we consider three different problems in the field of combinatorial optimization including "One-dimensional Bin Packing" (and two similar problems), "Machine Reassignment" and "Rolling Stock unit management on railway sites". The first one is a classical and well known optimization problem, while the other two are real world and very large scale problems arising in industry and have been recently proposed by Google and French Railways (SNCF) respectively. For each problem we propose a local search based heuristic algorithm and we compare our results with the best known results in the literature. Additionally, as an introduction to local search methods, two metaheuristic approaches, GRASP and Tabu Search are explained through a computational study on Set Covering Problem.
**Keywords:** Combinatorial optimization, Local search, Metaheuristics

**Résumé** Cette thèse porte sur la conception et l'implémentation d'algorithmes approchés pour l'optimisation en variables discrètes. Plus particulièrement, dans cette étude nous nous intéressons à la résolution de trois problèmes combinatoires difficiles : le « Bin-Packing », la « Réaffectation de machines » et la « Gestion des rames sur les sites ferroviaires ». Le premier est un problème d'optimisation classique et bien connu, tandis que les deux autres, issus du monde industriel, ont été proposés respectivement par Google et par la SNCF. Pour chaque problème, nous proposons une approche heuristique basée sur la recherche locale et nous comparons nos résultats avec les meilleurs résultats connus dans la littérature. En outre, en guise d'introduction aux méthodes de recherche locale mise en œuvre dans cette thèse, deux métaheuristiques, GRASP et Recherche Tabou, sont présentées à travers leur application au problème de la couverture minimale.
**Mots clés:** Optimisation combinatoire, Recherche locale, Métaheuristiques