UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
BACHELOR OF COMPUTER SCIENCE

GABRIEL MARQUES PORTAL

An Algorithmic Study of the Machine Reassignment Problem

Final Report presented in partial fulfillment of the requirements for the degree of Bachelor of Computer Science

Prof. Marcus Rolf Peter Ritt
Advisor

Porto Alegre, June 2012

CIP – CATALOGING-IN-PUBLICATION

Portal, Gabriel Marques

An Algorithmic Study of the Machine Reassignment Problem / Gabriel Marques Portal. – Porto Alegre: UFRGS, 2012.

48 f.: il.

Final Report (Bachelor) – Universidade Federal do Rio Grande do Sul. Bachelor of Computer Science, Porto Alegre, BR–RS, 2012. Advisor: Marcus Rolf Peter Ritt.

1. Heuristic algorithms. 2. Scheduling. 3. Resource optimization.

CONTENTS

LIST OF FIGURES 5

LIST OF TABLES 6

ABSTRACT 7

RESUMO 8

1 INTRODUCTION 9

2 PROBLEM DEFINITION 11
2.1 Context 11
2.2 Problem Elements 11
2.3 Hard Constraints 13
2.3.1 Capacity Constraints 13
2.3.2 Conflict Constraints 13
2.3.3 Dependency Constraints 14
2.3.4 Transient Usage Constraints 14
2.4 Objective Function 14
2.4.1 Load Cost 15
2.4.2 Balance Cost 15
2.4.3 Process Move Cost 16
2.4.4 Machine Move Cost 16
2.4.5 Service Move Cost 16
2.4.6 Total Objective Cost 16
2.5 Numerical Example 16
2.5.1 Instance Data 16
2.5.2 Original Assignment 18
2.5.3 New Assignment 19
2.6 Instances 19

3 INTEGER PROGRAMMING FORMULATION 21
3.1 Formulation 21
3.1.1 Decision Variables 21
3.1.2 Constraints 22
3.1.3 Objective Function 23
3.2 Results 24

4 LOWER BOUND 26
4.1 Load Cost Lower Bound 26
4.2 Balance Cost Lower Bound 27
4.3 Combined Lower Bound 28

5 LOCAL SEARCH METHODS 31
5.1 Randomized Local Search 31
5.1.1 Neighborhoods 31
5.1.2 Data Structures 31
5.1.3 Local Search Procedure 33
5.1.4 Results 34
5.2 Simulated Annealing 35
5.2.1 Algorithm 37
5.2.2 Parameter Setting 37
5.2.3 Results 39

6 SERVICE ROTATION 41
6.1 Insight and Modeling 41
6.2 Results 42

7 TWO-MACHINE NEIGHBORHOOD 44
7.1 Algorithm 44
7.2 Results 45

8 CONCLUSION 47

REFERENCES 48

LIST OF FIGURES

Figure 2.1: Usage of the machines in the original assignment. 18

Figure 6.1: Assignment Problem for the service rotation. 42

LIST OF TABLES

Table 2.1: Informations about the machines. 17
Table 2.2: Informations about the processes. 17
Table 2.3: Machine move costs. 17
Table 2.4: Information about instances. 20

Table 3.1: CPLEX results. 25

Table 4.1: Lower bounds. 29
Table 4.2: Lower bound values. 30

Table 5.1: Comparison between First Improvement and Best Improvement heuristics. 35
Table 5.2: Improved local search algorithms results. 36
Table 5.3: Results of the Simulated Annealing on instances A. 40
Table 5.4: Results of the Simulated Annealing on instances B. 40

Table 6.1: Results using Service Rotation. 43

Table 7.1: Results for the Randomized Two Machine movements. 46

ABSTRACT

In this work, an approach to solve the Machine Reassignment Problem is proposed. The problem consists in optimizing the usage of computational resources given a set of processes that must be executed. Therefore, an assignment of processes to machines must be find. Besides, there are plenty of constraints for increasing safety and reliability of the system. The problem has an important practical relevance nowadays. The increasing usage of cloud computing concepts forces the resource optimization of these huge computational infrastructures.

This is a very complex NP-Hard combinatorial optimization problem. Thus, the choice of an heuristic method is natural for solving the problem approximately. The proposed solution is an heuristic based on Simulated Annealing that uses two neighborhoods. The results are satisfactory since near-optimal solutions are found in less than five minutes.

Besides the proposed algorithm, this work presents some other contributions. An integer programming formulation is proposed and implemented, being capable of finding optimal solutions for small instances of the problem. To evaluate the solutions found by the proposed method for big instances, a lower bound was developed. The lower bounds seem to be good approximations of the optimal solutions in most cases.

The Machine Reassignment Problem was proposed in the context of the ROADEF / EURO Challenge, an international challenge proposed by the French Society of Operations Research. The presented problems always are important in Industry. This year, Google proposed the problem, interested in methods for optimizing its computational infrastructure. Among 83 participating teams, 30 have been classified for the final phase, and our team (representing UFRGS) ranked fourth with a preliminary version of the proposed method. The final results are not yet published as of the publication date of this work.

Keywords: Heuristic algorithms, scheduling, resource optimization.

Estudo Algorítmico do Problema de Reatribuição de Máquinas

RESUMO

O trabalho realizado propõe uma abordagem para resolver o problema de Reatribuição de Máquinas (*Machine Reassignment Problem*). O problema consiste em otimizar o uso de recursos computacionais dado um conjunto de processos que devem ser executados. Portanto, deve-se encontrar uma atribuição de processos a máquinas. Além disso, existem diversas restrições para aumentar a segurança e confiabilidade do sistema. O problema tem uma relevância prática bastante grande atualmente. Com um uso cada vez maior de computação nas nuvens (*Cloud Computing*), é importante a otimização dos recursos dessas grandes infra-estruturas computacionais.

Este é um problema de otimização combinatorial NO-Difícil muito complexo. Portanto, é natural a escolha de um método heurístico para resolvê-lo de forma aproximada. A solução proposta é uma heurística baseada em *Simulated Annealing* que utiliza duas vizinhanças. Os resultados são bem satisfatórios, visto que soluções quase-ótimas são encontradas em pouco tempo.

Além do método de solução proposto, obteve-se outras constribuições no decorrer do trabalho. Uma formulação com programação inteira foi proposta e implementada, sendo capaz de encontrar soluções ótimas para instâncias pequenas. Para avaliar a solução encontrada pelo método para instâncias grandes, foram desenvolvidos limitantes inferiores para a solução ótima. Os limitantes inferiores encontrados parecem ser bons comparados com a solução ótima na maioria dos casos.

O problema de Reatribuição de Máquinas foi proposto no contexto do desafio ROADEF, um desafio internacional proposto pela Sociedade Francesa de Pesquisa Operacional. O problema apresentado para ser resolvido é sempre um problema com relevância no meio industrial. Este ano, quem propôs o problema foi a empresa Google, interessada em métodos para otimização de sua infra-estrutura computacional. Dentre as 83 equipes participantes, 30 classificaram-se para a fase final e a nossa equipe (representando a UFRGS) ficou em quarto lugar com uma versão preliminar do método proposto. Os resultados finais ainda não haviam sido publicados na data de publicação deste trabalho.

Palavras-chave: algoritmos heurísticos, escalonamento, otimização de recursos.

1 INTRODUCTION

In the context of huge service providers like Google and Amazon, there is a big infrastructure to be dealt with. A lot of resources are purchased in order to provide a high quality service to the client. It is natural that these service providers have a great interest in optimizing the usage of their resources, otherwise more resources would have to be purchased to compensate this non-optimal usage. One of the main resources to be optimized in the technology environment is the computer power.

It is important to maximize the utilization of given resources, however there are other factors that also must be considered in this context. For example, it is necessary to guarantee the robustness of the system. The impact of a machine failure must be minimized and even if a whole data center shuts down it is desired to keep the system running. Also, an overload of the system increases the probability of failure and should be avoided. So, the goal is to optimize the resources keeping a high reliability.

This is a timely subject because of the increase of cloud computing services. It means that computing and storage capacity are being offered as services in the Internet. End users access all their information using a terminal with Internet connection. That shows the importance of the reliability of the system: in case of a breakdown, users may lose access to their data.

This work approaches a combinatorial optimization problem called Machine Reassignment Problem. This problem tries to model the main characteristics of this practical problem, even though this is not an easy task. There is a set of machines: the available resources. There is also a set of processes responsible for providing one or more services. The objective is to find an assignment of processes to machines that optimizes the usage of the machines while still guaranteeing the reliability of the system.

This problem is proposed in the context of the ROADEF/EURO challenge 2012. The ROADEF challenge is in its eighth edition. It is organized in partnership between an industrial enterprise and the french society of researches in operational research (ROADEF - *la Société Française de Recherche Opérationnelle et d'Aide à la Décision*). The challenge has already counted with the partnership of Renault, France Telecom, EDF and Amadeus. This time, the industrial partner is Google. The challenge is being organized jointly with the European Operational Research Society (EURO) in this edition. In the words of the organizers:

> The goal of this challenge is twofold. On the one hand, it allows some of our industrial partners to witness recent developments in the field

> of Operations Research and Decision Analysis, and young researchers to face up to a complex decisional problem occurred in industry. The challenge will give them an opportunity to explore the requirements and difficulties encountered in industrial applications. On the other hand, we hope that this challenge will help to establish a permanent partnership between manufacturers and young scientists on industrial size projects which require both high scientific qualification and the real-life practices in companies making use of decision analysis.
>
> (ROADEF [2009])

The subject of the challenge was provided in 8^{th} June of 2011, when the challenge effectively began. The competitors had until 8^{th} December of 2011 to send their methods for the qualification phase. Eighty three (83) teams all over the world registered and submitted their programs, but only thirty (30) teams qualified for the next phase. Our team placed 4^{th} in this preliminary ranking. Then, the teams had until 8^{th} June of 2012 to send their final methods. The final results will be published in 8^{th} July of 2012 in Vilnius (Lithuania), during the EURO conference. So, in the moment of finalizing this work, we do not know the final results of the competition.

The team representing the UFRGS (*Universidade Federal do Rio Grande do Sul*) was composed by Gabriel Portal, Marcus Ritt, Luciana Buriol, Leonardo Borba and Alexander Benavides.

2 PROBLEM DEFINITION

In this section, the Machine Reassignment Problem will be defined. First, the context of the problem will be exposed. Next, the main elements that compose the problem will be presented. The hard and soft restrictions, as well as the objective function, are then defined formally. At the end of the chapter, a small example is constructed step-by-step.

2.1 Context

In the Machine Reassignment Problem, the main elements are a set of machines and a set of processes. There is also an initial assignment of processes to machines. The objective is to improve the machine usage by finding a better assignment

The practical utility of the problem is very easy to understand. Google, who proposed the problem definition, has thousands of machines spread over the world, each of them receiving a huge number of requests every day due to all the offered services. It is natural to try to optimize the usage of the available resources. The concept of optimizing resources will be clearer with the definition of the objective function.

Although the proposed name of the problem is Machine Reassignment, it seems more natural (and that is how the problem will be referenced in this work) that processes are assigned to machines and not the other way. That is mainly because the relation assignment of processes to machines is functional: each process is assigned to one single machine at a time. However, the opposite is not true: a machine usually has many processes assigned to itself.

For the original definition of the problem, we refer the reader to ROADEF/Google [2011].

2.2 Problem Elements

In this section, the elements of the problem will be presented. They will be useful for later defining the constraints and the objective function. As it was previously stated, the main elements of the problem are a set of machines $\mathcal{M}$ and a set of processes $\mathcal{P}$. An assignment of processes to machines, a solution to the problem, is the definition of a function $A : \mathcal{P} \mapsto \mathcal{M}$ that maps every process $p \in \mathcal{P}$ to a machine $m \in \mathcal{M}$. Then, $A(p) = m$ means that process p is assigned to machine m in solution A. The function A_0 will represent the initial assignment of processes to machines.

Furthermore, each machine has available resources. For example, the processing capacity (CPU) and memory (RAM) are two important resources in the context of computers. Physical storage capacity (HD) could be a considered resource too. These are the most "classical". More practical "resources" may be: the number of simultaneous network connections, the network bandwidth, the number of simultaneous processes or threads, the number of open files, etc. As each machine has a quantity of each resource available, each process has an associated requirement for each resource. It will be natural that a constraint of the problem will be to respect the resource capacities of each machine. Formally, let $\mathcal{R}$ be the set of resources, $C(m, r)$ be the capacity of machine $m \in \mathcal{M}$ for resource $r \in \mathcal{R}$ and $R(p, r)$ be the requirement of process $p \in \mathcal{P}$ for resource $r \in \mathcal{R}$.

A resource may have transient usage costs. A resource that needs transient usage occupies space on the origin and destination machines while being transfered. A good example is the disk space (HD): when moving a process from one machine to another, disk space will be used (or reserved) in both machines. Formally, $\mathcal{TR} \subseteq \mathcal{R}$ is the subset of the resources that need transient usage.

Processes are partitioned into services. This means that each process is associated to a single service. Let $\mathcal{S}$ be the set of services, then $\forall p \in \mathcal{P}, \exists s \in \mathcal{S} \mid p \in s$. The semantics associated with services is a set of processes that have a common objective. In the context of Google, examples would be the email service GMail or the localization service GMaps. There are many processes, spread over many machines, that act to deliver a service of quality to the client.

Machines are partitioned in locations. Let $\mathcal{L}$ be the set of locations, then $\forall m \in \mathcal{M}, \exists l \in \mathcal{L} \mid m \in l$. Locations could mean geographical positioning. Considering that Google has many clusters of machines spread over the world, each cluster could represent a location (set of machines).

Machines are also partitioned in neighborhoods. Let $\mathcal{N}$ be the set of neighborhoods, then $\forall m \in \mathcal{M}, \exists n \in \mathcal{N} \mid m \in n$. The semantics associated with neighborhoods is the ease of communication. Two machines will be in the same neighborhood if they can exchange data fast enough to answer the requests of an user in acceptable time. Intuitively, neighborhoods would be a refinement of locations: the answer time is so small that two machines must be in the same geographical location to cooperate (and this is not always enough). Another possibility is that locations are a refinement of neighborhoods, assuming a more flexible response time for requests. So, a group of locations are sufficiently near to cooperate with each other. The formal definition of the problem does not make any of these assumptions. Therefore, locations and neighborhoods are independent partitions of the set of machines.

It is important to note that the semantical interpretations of the elements of the problem are, in majority, assumptions of the author of this work. Possibly, these assumptions do not reflect the ideas of the creators of the problem. It is also possible that the assumptions in relation to the infrastructure of the enterprise Google (used as example) are not correct.

2.3 Hard Constraints

Hard constraints must be respected by any valid solution for the problem. In contrast, the soft constraints are penalized by a cost in the objective function when they are violated. In this section, the five hard constraints that compose the Machine Reassignment Problem will be explained:

- capacity constraints,
- conflict constraints,
- spread constraints,
- dependency constraints and
- transient usage constraints.

2.3.1 Capacity Constraints

Capacity constraints guarantee that the usage of the resources of a machine does not exceed its capacity. These are the most intuitive constraints in the sense that it is not possible to use resources that do not exist – resources not available in the machine. So, it is very natural that it is a hard constraint.

Let $U(m, r)$ be the usage of resource $r \in \mathcal{R}$ in machine $m \in \mathcal{M}$. Considering an assignment function A, we have

$$U(m, r) = \sum_{p \in \mathcal{P} | A(p) = m} R(p, r). \tag{2.1}$$

Therefore, the capacity constraints are

$$U(m, r) \leq C(m, r),\ \forall m \in \mathcal{M},\ \forall r \in \mathcal{R}. \tag{2.2}$$

2.3.2 Conflict Constraints

Conflict constraints guarantee that no two processes of the same service on the same machine. This constraint aims to minimize the trouble caused to a single service in case of failure of a machine. In this case only one process of each service will be affected. The impact is spread more evenly among the services.

Therefore, conflict constraints are as follow:

$$A(p_i) \neq A(p_j),\ \forall s \in \mathcal{S},\ p_i \in s,\ p_j \in s. \tag{2.3}$$

2.3.2.1 Spread Constraints

Spread constraints guarantee that the processes of a service will be sufficiently spread among the machine locations, i.e. geographically dispersed. So, in case of failure of all the machines of a location, the services will still be able to answer their requests. Failures in all the machines of a region could be due to a natural catastrophe or even focused attacks (in case of war, or even elimination of sensible information).

Let spread_s, $s \in \mathcal{S}$, be the minimal number of locations that must be occupied by processes of service s. Let $L(P')$ be the set of locations occupied by processes in P', where $P' \subseteq \mathcal{P}$.

$$L(P') = \{l \in L \mid \exists m \in l, p \in P' : A(p) = m\}. \tag{2.4}$$

The definition of the spread constraints is

$$|L(s)| \geq spread_s, \; \forall s \in \mathcal{S}. \tag{2.5}$$

2.3.3 Dependency Constraints

Dependency constraints guarantee the efficiency of communication between services. Naturally, the neighborhoods will be involved in the definition of these constraints. Let service s_i depend on service s_j, there must be at least one process of service s_j in the same neighborhood of each process of service s_i. This definition implies that any process of service s_j may answer to the information requests of service s_i.

Formally, $s_i \Rightarrow s_j$ indicates that service s_i depends on service s_j. This way, we have the following constraints:

$$\forall p_i \in s_i, \; \exists p_j \in s_j, \; \exists n \in \mathcal{N} \mid A(p_i) \in n \wedge A(p_j) \in n \; (\forall s_i \Rightarrow s_j). \tag{2.6}$$

2.3.4 Transient Usage Constraints

Transient usage constraints complement the capacity constraints. Since some resources consume twice its requirements during the migration between machines, this extra usage of resources must be taken into account. For example, it is not possible to free the disk space on the origin machine before the content is copied to the target machine, however this space must be already reserved there.

So, capacity constraints of transient usage resources will be slightly different from Equation 2.2, but the former inequalities still apply. In this new formulation, the sum of requirements of processes originally assigned to machine $m \in \mathcal{M}$ or being assigned to m in the new solution must be considered for verification of capacity. We have

$$\sum_{p \in \mathcal{P} \mid A_0(p) = m \vee A(p) = m} R(p, r) \leq C(m, r), \; \forall m \in \mathcal{M}, \; r \in \mathcal{TR} \tag{2.7}$$

2.4 Objective Function

In this section, it will be described how to evaluate a solution. The elements of the objective function may be called soft constraints, since they define desirable characteristics of a solution but are not mandatory. The objective function is composed of the following components:

- load cost,
- balance cost,
- move cost

- process move cost
- machine move cost
- service move cost

2.4.1 Load Cost

Load cost is the term of the objective function responsible for avoiding overload of the machines. Even though a machine has a maximum capacity for a resource, it is desirable to keep the usage of this resource lower for a better operation of the machine. For example, if the memory resource of a machine is completely used, any non-usual operation (e.g. for the operational system) would exceed the available capacity and compromise the performance of the machine. This constraint formalizes the desire to keep a security margin over the capacity of the resources, even though this is not mandatory.

Let $SC(m, r)$ be the safety capacity of machine $m \in \mathcal{M}$ for resource $r \in \mathcal{R}$. It is desired that the usage of the resource r in machine m to be less than $SC(m, r)$, otherwise it will be paid linearly for the overload. Note that a usage exceeding $C(m, r)$ is not allowed. Therefore, it makes sense that $C(m, r) \geq SC(m, r)$. The load cost for a given resource is stated as

$$\text{load}(r) = \sum_{m \in \mathcal{M}} \max(0, U(m, r) - SC(m, r)). \tag{2.8}$$

Let $\text{weight}_{\text{lc}}(r)$ be the weight of the load cost for resource r. The load cost over all the resources is

$$\text{loadCost} = \sum_{r \in \mathcal{R}} \text{weight}_{\text{lc}}(r)\ \text{load}(r). \tag{2.9}$$

2.4.2 Balance Cost

Balance cost is the term of the objective function responsible for balancing resources in the machine. The balance of resources is especially important for future usage of the machine. It is pointless, for example, to have free memory space on a machine if there are no processing resources available. This constraint states a kind of "dependency" between resources: if a machine has x available units of resource r_i, it needs to have at least tx available units of resource r_j, for a constant t.

Let $\mathcal{B}$ be the set of dependencies between resources. This dependency is defined by a pair of resources and a target ratio. Formally, we have $b = (r_1, r_2, t) \in \mathcal{B}$ and $\mathcal{B} \subseteq \mathcal{R}^2 \times \mathbb{N}$. We will use projection functions to access the elements of a triple. So, $T(b)$, $R_1(b)$ and $R_2(b)$ represent the elements of the balance cost b. Let $F(m, r)$ be the free space of resource $r \in \mathcal{R}$ in machine $m \in \mathcal{M}$. Then

$$F(m, r) = C(m, r) - U(m, r). \tag{2.10}$$

The balance cost for a given $b \in \mathcal{B}$ is

$$\text{balance}(b) = \sum_{m \in \mathcal{M}} \max(0, T(b)\ F(m, R_1(b)) - F(m, R_2(b))). \tag{2.11}$$

Let $\text{weight}_{\text{bc}}(b)$ be the weight of balance cost b. Then, the total balance cost is

$$\text{balancecost} = \sum_{b \in \mathcal{R}} \text{weight}_{\text{bc}}(b)\ \text{balance}(b). \tag{2.12}$$

2.4.3 Process Move Cost

Moving processes around is undesirable because they will not answer requests during the transfer time. Besides, some processes are more costly to move than others, due to the amount of data to be transfered, for example. Let $\mathrm{PMC}(p)$ be the move cost of process $p \in \mathcal{P}$ and $\mathrm{weight}_{\mathrm{pmc}}$ be the weight of the process move cost. Then, the total process move cost is

$$\mathrm{processMoveCost} = \mathrm{weight}_{\mathrm{pmc}} \sum_{p \in \mathcal{P} | A(p) \neq A_0(p)} \mathrm{PMC}(p). \tag{2.13}$$

2.4.4 Machine Move Cost

Moving a process between machines that are in the same room may be different from moving a process between machines in different continents. At least, the network speed will be different, thus the transfer time will also be different. Let $MMC(m_i, m_j)$ be the cost of transferring a process from machine m_i to machine m_j. Naturally, $\mathrm{MMC}_{m,m} = 0$ for every machine $m \in \mathcal{M}$. Also, let $\mathrm{weight}_{\mathrm{mmc}}$ be the weight of the machine move cost. The total machine move cost is

$$\mathrm{machineMoveCost} = \mathrm{weight}_{\mathrm{mmc}} \sum_{p \in \mathcal{P}} \mathrm{MMC}_{A_0(p), A(p)}. \tag{2.14}$$

2.4.5 Service Move Cost

Service move cost measures how the services were affected by the transfer of processes for the new assignment. This cost reflects the maximum number of moved processes of a service, over all services. Let $\mathrm{MS}(s)$ be the set of processes of service s that moved out from their original machines. So,

$$\mathrm{MS}(s) = \{p \in s \mid A(p) \neq A_0(p)\}. \tag{2.15}$$

Let $\mathrm{weight}_{\mathrm{smc}}$ be the weight of the service move cost. The service move cost will be

$$\mathrm{serviceMoveCost} = \mathrm{weight}_{\mathrm{smc}} \sum_{s \in \mathcal{S}} |MS(s)|. \tag{2.16}$$

2.4.6 Total Objective Cost

After defining all the components of the objective function, the definition of the total objective function is the sum of all these terms. Therefore,

$$\begin{aligned} \mathrm{totalCost} = {} & \mathrm{loadCost} + \mathrm{balanceCost} + \\ & \mathrm{processMoveCost} + \mathrm{machineMoveCost} + \mathrm{serviceMoveCost}. \end{aligned} \tag{2.17}$$

2.5 Numerical Example

In this section an instance will be presented with the objective of illustrating many aspects of the problem. All the data will be shown and a new solution will be proposed.

2.5.1 Instance Data

In our working example, there will be 3 machines and 7 processes. So, $\mathcal{M} = \{m_1, m_2, m_3\}$ and $\mathcal{P} = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$. There are only two resources $\mathcal{R} = \{r_1, r_2\}$

Table 2.1: Informations about the machines.

Machine	r_1	r_2	Location	Neighborhood
m_1	16 (5)	170 (50)	l_1	n_1
m_2	9 (8)	150 (30)	l_1	n_2
m_3	17 (13)	140 (70)	l_2	n_2

Table 2.2: Informations about the processes.

Process	r_1	r_2	PMC	Service	A_0
p_1	9	60	15	s_1	m_1
p_2	5	20	7	s_2	m_1
p_3	4	10	5	s_2	m_2
p_4	2	50	7	s_4	m_2
p_5	2	40	6	s_3	m_2
p_6	3	10	4	s_3	m_3
p_7	3	20	5	s_1	m_3

and one of them is a transient usage resource $\mathcal{TR} = \{r_2\}$. Table 2.1 presents the capacity of each machine for each resource (safety capacities are given in parentheses), as well as the location and neighborhood of each machine. Table 2.2 presents the requirements of each process for each resource, as well as process move cost, original machine and service it belongs to. Finally, Table 2.3 presents the machine move costs.

Regarding Table 2.1, we can see there are two locations $\mathcal{L} = \{l_1, l_2\}$ and two neighborhoods $\mathcal{N} = \{n_1, n_2\}$. Namely, $l_1 = \{m_1, m_2\}$, $l_2 = \{m_3\}$, $n_1 = \{m_1\}$ and $n_2 = \{m_2, m_3\}$. Table 2.2 shows there are four services $\mathcal{S} = \{s_1, s_2, s_3, s_4\}$ and we have $s_1 = \{p_1, p_7\}$,$s_2 = \{p_2, p_3\}$,$s_3 = \{p_5, p_6\}$ and $s_4 = \{p_4\}$. The spread values for each of these services is 2, 1, 1, 1, respectively.

There is only one dependency: service s_2 depends on service s_1, $s_2 \Rightarrow s_1$. There is also only one balance triple $\mathcal{B} = \{b_1\}$ and $b_1 = \langle r_1, r_2, 10\rangle$. The weights are $\text{weight}_{\text{lc}}(r_1) = 90$, $\text{weight}_{\text{lc}}(r_2) = 10$, $\text{weight}_{\text{bc}}(b_1) = 5$ and the other weights are all 1.

Table 2.3: Machine move costs.

x	m_1	m_2	m_3
m_1	0	25	125
m_2	25	0	5
m_3	125	5	0

2.5.2 Original Assignment

First, we will analyze the original assignment to confirm that it respects all the constraints. The usage of the resources of the machines are as follows: $U(m_1, r_1) = 14$, $U(m_1, r_2) = 80$, $U(m_2, r_1) = 8$, $U(m_2, r_2) = 100$, $U(m_3, r_1) = 6$ and $U(m_3, r_2) = 30$. These values are less than the stated capacities in Table 2.1. Figure 2.1 illustrates the usage of the machines.

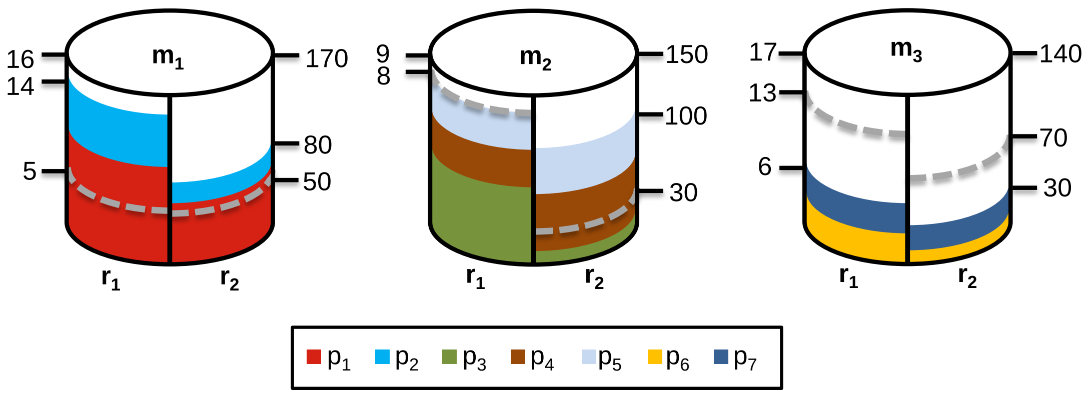

Figure 2.1: Usage of the machines in the original assignment.

The conflict constraint is also respected because the processes of a service are always in a different machine. A value 1 of minimum spread is trivially respected because a process of a service will always be assigned to some machine, and this machine must be in some location, resulting in a spread of at least 1. The only non-trivial spread constraint is for service s_1, which is respected since p_1 is on machine m_1 (location l_1) and p_7 is on machine m_7 (location l_2).

The only dependency constraint is the following: service s_2 depends on service s_1. The processes of s_2 are present in neighborhoods n_1 and n_2, so there must be at least a process of service s_1 in each of these neighborhoods. Process p_1 is in neighborhood n_1 and process p_7 is in neighborhood n_2, respecting the constraint. Finally, the transient usage constraint is trivially respected because no process was moved.

All hard constraints were verified, and now we will calculate the cost of the initial assignment. For resource r_1, we have $\text{load}(r_1) = 9 + 0 + 0 = 9$ and for resource resource r_2, $\text{load}(r_2) = 30 + 70 + 0 = 100$. Then, the total load cost is $\text{loadCost} = 90 \times 9 + 10 \times 100 = 1810$. For the balance cost, we must first calculate the free resources in the machines: $F(m_1, r_1) = 2$, $F(m_1, r_2) = 90$, $F(m_2, r_1) = 1$, $F(m_2, r_2) = 50$, $F(m_3, r_1) = 11$ and $F(m_3, r_2) = 110$. The target ratio of b_1 is 10, but after multiplying the usage of resource r_1 by 10 the values are still smaller than the free space of resource r_2. This way, the terms would be negative and the operator $\max$ sets the total balance cost to 0. There is obviously no move cost, since no process was moved. The total value of this solution is 1810.

2.5.3 New Assignment

The new proposed assignment is the following: $A(p_1) = m_3$, $A(p_2) = m_2$, $A(p_3) = m_3$, $A(p_4) = m_1$, $A(p_5) = m_3$, $A(p_6) = m_2$ and $A(p_7) = m_1$. First, it is necessary to check if the hard constraints are being respected. The usage of the machines are $U(m_1, r_1) = 5$, $U(m_1, r_2) = 70$, $U(m_2, r_1) = 8$, $U(m_2, r_2) = 30$, $U(m_3, r_1) = 15$ and $U(m_3, r_2) = 110$. It is clear that these values are below the capacity constraints. The conflict constraint is also respected, since processes in the same machine do not belong to the same service.

The spread constraints can be easily verified: processes of service s_1 are spread over machines from locations l_1 and l_2. With regard to the dependency constraint, both processes from service s_2 are in neighborhood n_2, so it is enough to exist a process of service s_1 in this neighborhood. Indeed, process p_7 from service s_1 is neighborhood n_2.

All the processes were moved in this new assignment, so the transient usage constraint must be verified for resource r_2. For machine m_1, processes p_1, p_2, p_4 and p_7 were there originally or are there in the new assignment, accumulating an use of 150 in resource r_2 (below the capacity). For machine m_2, the sum of requirements over all concerned processes is 130, also below the capacity. And for machine m_3, the sum of the requirements is 100, which is below 140.

Since all hard constraints are respected, this is a valid assignment. Then, the value of this assignment will be calculated. $\text{load}(r_1) = 2$ and $\text{load}(r_2) = 60$, resulting in $\text{loadCost} = 780$. The free space in the machines is $F(m_1, r_1) = 11$, $F(m_1, r_2) = 100$, $F(m_2, r_1) = 1$, $F(m_2, r_2) = 120$, $F(m_3, r_1) = 2$ and $F(m_3, r_2) = 30$. The only not balanced machine according to b_1 is m_1 because $T(b)\ F(m_1, r_1) \geq F(m_1, r_2)$. So, $\text{balance}(b_1) = 10$ and $\text{balanceCost} = 50$. All processes were moved, so the $\text{process MoveCost} = 49$, the $\text{machineMoveCost} = 315$ and the $\text{serviceMoveCost} = 2$. Summing all these values, we have $\text{totalCost} = 1196$.

2.6 Instances

During the ROADEF Challenge 2011/2012, 20 instances were made available. Before the qualification phase, ten A instances were given to the competitors to test their methods. After the qualification phase, another 10 larger instances were given. Table 2.4 shows some information about these instances, which we used during the development of this work. The table shows the number of processes ($|P|$), the number of machines ($|M|$), the number of resources ($|R|$), the number of services ($|S|$), the number of locations ($|L|$), the number of neighborhoods ($|N|$), the number of dependencies (dep.), and the number of balance costs ($|B|$) of each instance. It is also presented the value of the objective function for the original assignment (Initial Solution).

It is worth noting that the number of locations and neighborhoods is the same as the number of machines in all instances. Together with the conflict constraint, it means that every process of a service will be in a different location and a different neighborhood. As a direct consequence, the spread constraint is trivially satisfied in all these instances.

Table 2.4: Information about instances.

Instance	$\|\mathcal{P}\|$	$\|\mathcal{M}\|$	$\|\mathcal{R}\|$	$\|\mathcal{S}\|$	$\|\mathcal{L}\|$	$\|\mathcal{N}\|$	dep.	$\|\mathcal{B}\|$	Initial Solution
A1-1	100	4	2	79	4	4	0	1	49528750
A1-2	1000	100	4	980	100	100	40	0	1061649570
A1-3	1000	100	3	216	100	100	342	0	583662270
A1-4	1000	50	3	142	50	50	297	1	632499600
A1-5	1000	12	4	981	12	12	32	1	782189690
A2-1	1000	100	3	1000	100	100	0	0	391189190
A2-2	1000	100	12	170	100	100	0	0	1876768120
A2-3	1000	100	12	129	100	100	577	0	2272487840
A2-4	1000	50	12	180	50	50	397	1	3223516130
A2-5	1000	50	12	153	50	50	506	1	787355300
B-1	5000	100	12	2512	100	100	4412	0	7644173180
B-2	5000	100	12	2462	100	100	3617	1	5181493830
B-3	20000	100	6	15025	100	100	16560	0	6336834660
B-4	20000	500	6	1732	500	500	40485	1	9209576380
B-5	40000	100	6	35082	100	100	14515	0	12426813010
B-6	40000	200	6	14680	200	200	42081	1	12749861240
B-7	40000	4000	6	15050	4000	4000	43873	1	37946901700
B-8	50000	100	3	45030	100	100	15145	0	14068207250
B-9	50000	1000	3	4609	1000	1000	43437	1	23234641520
B-10	50000	5000	3	4896	5000	5000	47260	1	42220868760

3 INTEGER PROGRAMMING FORMULATION

An Integer Programming (IP) formulation is a mathematical formulation of a problem using integer variables. The stated constraints (or inequalities) and the objective function must be linear.

It is a good practice to state the problem in a mathematical form to be sure that there are no ambiguities or misunderstandings in its definition. An integer programming formulation is an interesting option even being a little restrictive (due to linearity constraints) because there are optimized solvers available. This way, the IP solver is also a standard parameter to compare methods against.

In this section, an integer programming formulation for the Machine Reassignment Problem will be presented. This formulation was implemented in the CPLEX solver and the results are also presented.

3.1 Formulation

3.1.1 Decision Variables

- $x_{pm} \in \{0,1\}$ indicates the assignment of process $p \in \mathcal{P}$ to machine $m \in \mathcal{M}$.
- $y_p \in \{0,1\}$ indicates whether the process $p \in \mathcal{P}$ changed from its original machine.
- $U_{mr} \in \mathbb{R}_+$ indicates the usage of machine $m \in \mathcal{M}$ for resource $r \in \mathcal{R}$.
- $A_{sl} \in \{0,1\}$ indicates whether service $s \in \mathcal{S}$ has a service in location $l \in \mathcal{L}$.
- $B_{sn} \in \{0,1\}$ indicates whether service $s \in \mathcal{S}$ has a service in neighborhood $n \in \mathcal{N}$.
- $\mathrm{load}_{mr} \in \mathbb{R}_+$ is the load cost in machine $m \in \mathcal{M}$ for resource $r \in \mathcal{R}$.
- $\mathrm{lcost} \in \mathbb{R}_+$ is the total load cost.
- $\mathrm{balance}_{bm} \in \mathbb{R}_+$ is the balance cost $b \in \mathcal{B}$ for machine $m \in \mathcal{M}$.
- $\mathrm{bcost} \in \mathbb{R}_+$ is the total balance cost.
- $\mathrm{pmcost} \in \mathbb{R}_+$ is the total process move cost.
- $\mathrm{service} \in \mathbb{R}_+$ is the service move cost.
- $\mathrm{smcost} \in \mathbb{R}_+$ is the total service move cost.

- $\text{machine}_p \in \mathbb{R}_+$ is the machine move cost for process $p \in \mathcal{P}$.
- $\text{mmcost} \in \mathbb{R}_+$ is the total machine move cost.
- $\text{totalcost} \in \mathbb{R}_+$ is the total cost, which must be minimized.

3.1.2 Constraints

We must assign each process to exactly one machine

$$\sum_{m \in \mathcal{M}} x_{pm} = 1 \qquad \forall p \in \mathcal{P}. \tag{3.1}$$

A process changed from its original machine if it is not assigned to it

$$y_p = 1 - x_{pA_0(p)} \qquad \forall p \in \mathcal{P}. \tag{3.2}$$

The usage of resource $r \in R$ on machine $m \in M$ is defined by

$$U_{mr} = \sum_{p \in \mathcal{P}} x_{pm} R(p, r), \tag{3.3}$$

and, therefore, the capacity constraints are

$$U_{mr} \leq C(m, r) \qquad \forall m \in \mathcal{M}, r \in \mathcal{R} \setminus \mathcal{TR}. \tag{3.4}$$

Note that only the non-transient resources are being considered, even though the constraint would also be correct for them. However stronger constraints will be defined for them.

The conflict constraints permit at most one process of a service on a machine. They are guaranteed by

$$\sum_{p \in s} x_{pm} \leq 1 \qquad \forall s \in \mathcal{S}, m \in \mathcal{M}.^1 \tag{3.5}$$

The definition of A_{sl} is guaranteed by the following constraints

$$A_{sl} \leq \sum_{m \in l, p \in s} x_{pm} \qquad \forall s \in \mathcal{S}, l \in \mathcal{L}. \tag{3.6}$$

Note that if there is no process from service s in location l, variable A_{sl} is forced to be 0. Otherwise, it is free to assume any value, however the optimization process will force it to be 1 if necessary. Together with

$$\sum_{l \in \mathcal{L}} A_{sl} \geq \text{spread}_s \qquad \forall s \in \mathcal{S}, \tag{3.7}$$

these constraints guarantee that spread constraints are being respected. To satisfy Equation 3.7, it is desirable that variables A_{sl} assume the value 1 when possible, and therefore these variables in conjunction with Equation 3.6 assume the right values.

[1]Constraints $x_{sm} + x_{tm} \leq 1$ for all $s \neq t \in \mathcal{S}$ and all $m \in \mathcal{M}$ are weaker.

The definition of B_{sl} is guaranteed by

$$B_{sn} \leq \sum_{m \in n, p \in s} x_{pm} \qquad \forall s \in \mathcal{S}, n \in \mathcal{N}. \tag{3.8}$$

Note that this variable is defined similar to variables A_{sl}. The same observations apply here and some other constraint will have to force these variables to be 1. These constraints are

$$B_{sn} \geq \sum_{m \in n, p \in s} x_{pm} / \min\{|n|, |s|\} \qquad \forall n \in \mathcal{N}, s \in \mathcal{S}. \tag{3.9}$$

If there is no process of service s in neighborhood n, the constraint is always satisfied. Otherwise, the right side of the inequality will be greater than 0 but at most 1 because the number of processes from service s in neighborhood n is at most $\min\{|n|, |s|\}$. Then, the constraint will force variable B_{sn} to be 1. Finally, the dependency constraints are guaranteed by

$$B_{sn} \leq B_{tn} \qquad \forall n \in \mathcal{N}, s \Rightarrow t. \tag{3.10}$$

Besides considering the usage of a machine, transient resources must consider the usage of resources in original machines. Transient usage constraints are, then, defined by

$$U_{mr} + \sum_{p \in P | A_0(p) = m} y_p R(p, r) \leq C(m, r) \qquad \forall m \in \mathcal{M}, r \in \mathcal{TR}. \tag{3.11}$$

3.1.3 Objective Function

The variable load_{mr} is defined by

$$\mathrm{load}_{mr} \geq U_{mr} - SC(m, r) \qquad \forall m \in \mathcal{M}, r \in \mathcal{R} \tag{3.12}$$

and the total load cost is

$$\mathrm{lcost} = \sum_{m \in \mathcal{M}, r \in \mathcal{R}} \mathrm{weight}_{\mathrm{lc}}(r)\, \mathrm{load}_{mr}. \tag{3.13}$$

Variables load_{mr} could be arbitrarily big by these constraints, but since they must be minimized by the objective function, an optimal solution will satisfy the restrictions with equality.

In a similar way, the balance cost is defined by

$$\mathrm{balance}_{bm} \geq T(b)(C(m, R_1(b)) - U_{m,R_1(b)}) - (C(m, R_2(b)) - U_{m,R_2(b)}), \tag{3.14}$$

then the total balance cost is

$$\mathrm{bcost} = \sum_{m \in \mathcal{M}, b = \in \mathcal{B}} \mathrm{weight}_{\mathrm{bc}}(b)\, \mathrm{balance}_{mb}. \tag{3.15}$$

The definition of process move cost is

$$\mathrm{pmcost} = \mathrm{weight}_{\mathrm{pmc}} \sum_{p \in \mathcal{P}} y_p\, \mathrm{PMC}(p). \tag{3.16}$$

The service move cost is constrained by

$$\text{service} \geq \sum_{p \in s} y_p \qquad \forall s \in \mathcal{S}. \tag{3.17}$$

Then the total service move cost is simply

$$\text{smcost} = \text{weight}_{\text{smc}}\, \text{service}. \tag{3.18}$$

The machine move cost for process $p \in \mathcal{P}$ is

$$\text{machine}_p = \sum_{m \in \mathcal{M}} x_{pm}\, \text{MMC}_{A_0(p),m} \qquad \forall p \in \mathcal{P}. \tag{3.19}$$

Then, the total machine move cost is

$$\text{mmcost} = \text{weight}_{\text{mmc}} \sum_{p \in \mathcal{P}} \text{machine}_p. \tag{3.20}$$

The total objective function is

$$\text{totalcost} = \text{lcost} + \text{bcost} + \text{pmcost} + \text{mmcost} + \text{smcost} \tag{3.21}$$

and the objective of the problem is

$$\min\ \text{totalcost}. \tag{3.22}$$

3.2 Results

The proposed formulation was implemented in the software CPLEX version 12.3. Basically, a branch-and-bound algorithm is used to solve the problem. However, CPLEX is a proprietary software and the details of the used techniques are not open to the community.

Instances A were solved by CPLEX using two threads and a maximum time limit of one hour. These results are shown in Table 3.1, in column “CPLEX [1h]”, as well as the gap from the internal lower bound found by CPLEX (column “$Gap1[\%]$”) and the spent time (column “time1 [s]”). The instances A-1, A1-3 and A1-5 were solved optimally, as can be observed by the gap, and also the execution ended before the stipulated limit. As some gaps were highly unsatisfactory, some instances were solved again with four threads and a time limit of 10 hours. These results are presented in the column “CPLEX2 [10h]” of the table, as well as its gap (column “$Gap2[\%]$”) and time (column “time2 [s]”). The quality of these solutions is much better, even though it demanded a computational effort 20 times higher. Note that a gap of 0% may not indicate an optimal solution since CPLEX informs the gap only to a precision of two digits.

The formulation of instances B for CPLEX could not fit in the main memory of the computer (12GB in the tested machine). Therefore, instances B were considered too big to be solved by integer programming.

Note that in the ROADEF challenge, the time limit to solve each instance was 300 seconds. This is a tight time limit considering the size of the instances and it was expected that the plain integer programming approach would not provide a competitive method. The objective, from the beginning, was to provide good bounds for the instances. This explains the extra computation time given to the CPLEX solver.

Table 3.1: CPLEX results.

Instance	CPLEX [1h]	$Gap1[\%]$	time1 [s]	CPLEX2 [10h]	$Gap2[\%]$	time2 [s]
A1-1	44306501	0.0	0.0	-	-	-
A1-2	778318293	0.1	3600.0	-	-	-
A1-3	583005925	0.0	25.4	-	-	-
A1-4	303303782	20.01	3600.0	272728697	7.91	36000
A1-5	727578313	0.0	4.9	-	-	-
A2-1	2350329	1298424.3	3600.0	181	0.0	36000
A2-2	1096122427	37.93	3600.0	794668498	0.0	36000
A2-3	1414426104	9.48	3600.0	-	-	-
A2-4	3018472741	79.62	3600.0	2040289999	21.41	36000
A2-5	706922741	129.85	3600.0	-	-	-

4 LOWER BOUND

Ideally, we would like to know the optimal solution for all the instances. That way, it is very easy to judge the quality of a method by the deviation of the found solutions from the optimal known solutions.

An useful approach is to determine lower bounds for the problem. A lower bound is a value that is certainly below (or equal) the optimal solution in a minimization problem. It should be as close as possible of the optimal solution because we will use this value to evaluate the quality of the solutions. Lower bounds are also very useful in exact methods, such as branch-and-bound. These methods search the entire solution space and good estimates of the optimal solution may result in pruning a great part of this space.

A lower bound is usually found by solving optimally a relaxed version of the problem. The most common relaxation for Integer Programming is the linear programming relaxation. The constraint that forces the variables to be integer is ignored and the problem can be solved by the Simplex method (or any other linear programming method). On the other hand, to find good problem specific lower bounds can be difficult.

In our context, the IP formulation for CPLEX was used in the beginning of the challenge, when the instances were not very large (instances A). CPLEX uses the linear programming relaxation of the problem, but this proved to be good enough for our initial evaluations.

When we had to work with the larger B instances, our IP formulation could not fit in the memory of the computer, i.e. that we could not determine a linear programming lower bound and should develop our own lower bound to evaluate our methods. This lower bound will be presented in this section.

4.1 Load Cost Lower Bound

Here, a lower bound for the load cost will be given. If we take, for every resource, the sum of the requirements over all processes and the sum of the safety capacities over all machines, the excess of resource requirements is a lower bound for the load cost. Let call it $\mathrm{LB}_{\mathrm{loadCost}}$, then

$$\mathrm{LB}_{\mathrm{loadCost}} = \sum_{r\in\mathcal{R}} \mathrm{weight}_{\mathrm{lc}}(r)\ \max(0, \sum_{p\in\mathcal{P}} R(p,r) - \sum_{m\in\mathcal{M}} SC(m,r)). \tag{4.1}$$

The proof of the validity of this lower bound is given by manipulating the definition of load cost. A valid identity that will be used here is the following:

$$\sum_{m\in\mathcal{M}} U(m,r) = \sum_{p\in\mathcal{P}} R(p,r),\ \forall r \in \mathcal{R}. \tag{4.2}$$

Another valid manipulation that will be used is that the sum of positive numbers of a sequence is greater than, or equal to, the maximum between 0 and the sum of the numbers of the same sequence. This will be useful to manipulate the $\max$ operator of the formulas.

$$\sum_{x\in X} \max(0,x) \geq \max(0, \sum_{x\in X} x) \tag{4.3}$$

Therefore, the proof proceeds like that:

$$\begin{aligned}\text{loadCost} = &\sum_{r\in\mathcal{R}} \text{weight}_{\text{lc}}(r) \sum_{m\in\mathcal{M}} \max(0, U(m,r) - SC(m,r)) \geq \\ &\sum_{r\in\mathcal{R}} \text{weight}_{\text{lc}}(r)\ \max(0, \sum_{m\in\mathcal{M}} (U(m,r) - SC(m,r))) = \\ &\sum_{r\in\mathcal{R}} \text{weight}_{\text{lc}}(r)\ \max(0, \sum_{m\in\mathcal{M}} U(m,r) - \sum_{m\in\mathcal{M}} SC(m,r)) = \\ &\sum_{r\in\mathcal{R}} \text{weight}_{\text{lc}}(r)\ \max(0, \sum_{p\in\mathcal{P}} R(p,r) - \sum_{m\in\mathcal{M}} SC(m,r))\end{aligned}$$

4.2 Balance Cost Lower Bound

A lower bound for the balance cost can be obtained in a similar way than the lower bound for the load cost. Instead of analyzing each machine separately for verifying the difference of free space for the considered resources, it is considered the free space for all machines and the requirements over all processes. Let call this lower bound $\text{LB}_{\text{balanceCost}}$, then we have

$$\text{LB}_{\text{balanceCost}} = \sum_{b\in\mathcal{B}} \text{weight}_{\text{bc}}(b)\ \max(0, T(b)\, E(R_1(b)) - E(R_2(b))), \tag{4.4}$$

where $E(r)$ is the excess of the total capacity for resource r over the total requirements for this resource, defined as

$$E(R) = \sum_{m\in\mathcal{M}} C(m,r) - \sum_{p\in\mathcal{P}} R(p,r). \tag{4.5}$$

To understand this lower bound, it is necessary to understand the meaning of the balance cost. Its objective is to balance the free space of resources in the machine and not the usage of resources. The proof of this lower bound is a manipulation of the definition of balance cost. In the following proof, we will use the identity

$$\begin{aligned}\sum_{m\in\mathcal{M}} F(m,r) = &\sum_{m\in\mathcal{M}} (C(m,r) - U(m,r)) = \\ &\sum_{m\in\mathcal{M}} C(m,r) - \sum_{m\in\mathcal{M}} U(m,r) = E(r)\end{aligned} \tag{4.6}$$

Then we have the following:

$$\begin{aligned}\text{balanceCost} = \sum_{b\in\mathcal{B}} \text{weight}_{\text{bc}}(b) \sum_{m\in\mathcal{M}} \max(0, T(b)\ F(m, R_1(b)) - F(m, R_2(b))) \geq \\ \sum_{b\in\mathcal{B}} \text{weight}_{\text{bc}}(b)\ \max(0, T(b) \sum_{m\in\mathcal{M}} F(m, R_1(b)) - \sum_{m\in\mathcal{M}} F(m, R_2(b))) = \\ \sum_{b\in\mathcal{B}} \text{weight}_{\text{bc}}(b)\ \max(0, T(b)\ E(R_1(b)) - E(R_2(b)))\end{aligned}$$

4.3 Combined Lower Bound

Since the load cost and balance cost enter separately into the objective function, we can combine them to obtain a lower bound for the problem:

$$\text{LB} = \text{LB}_{\text{loadCost}} + \text{LB}_{\text{balanceCost}}. \tag{4.7}$$

The derived lower bound might not be very strong. It ignores move costs and also consider load cost and balance cost independently. A tighter lower bound could be achieved solving a relaxed problem considering both costs.

Table 4.1 shows the lower bounds for all instances. For each instance, it shows the load cost lower bound ($\text{LB}_{\text{loadCost}}$), the balance cost lower bound ($\text{LB}_{\text{balanceCost}}$) and the final lower bound (lowerBound). It can be verified that for instances which have no balance cost (see Table 2.4), the $\text{LB}_{\text{balanceCost}}$ is naturally equal to 0.

Table 4.2 presents an evaluation of the proposed lower bound and a comparison with the lower bound generated by the linear programming relaxation of the problem. For each instance, it is presented the best known value (BKV), lower bound as presented in this section (lowerBound), the ratio of this lower bound in relation to the best known value (ratio1), the linear programming lower bound obtained by CPLEX (LP-LB), the ratio of this alternative lower bound (ratio2) and the time taken to generate this lower bound (time).

Evaluating first our own lower bound, we note there are many values close to 0% in column ratio1. This means that the lower bound is reasonably good in general. There are, however, some instances for which the lower bound is far from the best known value. Instances A2-2 and A2-3 have very high ratio1 values, indicating that the lower bound is probably far from the optimal solution. Besides, instance A2-1 has a lower bound of 0.

Comparing the lower bound with the linear programming relaxation (columns *LB* and *LP Lower Bound* of Table 4.2), we can see that LB values are always worse. This is expected since the linear relaxation deals with all elements of the problem, while our lower bound deals only with some elements and in an independent way. However, it is worth noting that the values are usually very close. Besides that, our lower bound is much simpler to calculate and it can be found in a very short time while this is not always true for linear relaxation (column time is there to show that some relaxations took a long time). Due to these factors, our lower bound allowed us to evaluate our methods with instances B while the linear programming relaxation could not be used because of memory and/or time constraints.

Table 4.1: Lower bounds.

Instance	$\mathrm{LB}_{\mathrm{loadCost}}$	$\mathrm{LB}_{\mathrm{balanceCost}}$	LB
A1-1	31011730	13294660	44306390
A1-2	777530730	0	777530730
A1-3	583005700	0	583005700
A1-4	0	242387530	242387530
A1-5	602301710	125276580	727578290
A2-1	0	0	0
A2-2	13590090	0	13590090
A2-3	521441700	0	521441700
A2-4	1450548890	229673490	1680222380
A2-5	307035180	0	307035180
B-1	3290754940	0	3290754940
B-2	31188860	983965000	1015153860
B-3	156631070	0	156631070
B-4	0	4677767120	4677767120
B-5	922858550	0	922858550
B-6	0	9525841820	9525841820
B-7	0	14833996360	14833996360
B-8	1214153440	0	1214153440
B-9	10050999350	5834370050	15885369400
B-10	0	18048006980	18048006980

Table 4.2: Lower bound values.

Instance	BKV	LB	ratio1[%]	LP-LB	ratio2[%]	time[s]
A1-1	44306501	44306390	0.00	44306481	0.00	0.0
A1-2	777532813	777530730	0.00	777530748	0.00	27.4
A1-3	583005717	583005700	0.00	583005701	0.00	23.4
A1-4	252728589	242387530	4.26	242394539	4.26	7.6
A1-5	727578309	727578290	0.00	727578296	0.00	0.2
A2-1	181	0	∞	66	174.24	2.5
A2-2	794668498	13590090	5747.41	29349216	2607.63	725.7
A2-3	1291984008	521441700	147.77	573189496	125.40	676.2
A2-4	1680487588	1680222380	0.01	1680230778	0.01	40.4
A2-5	307561267	307035180	0.17	307040661	0.17	28.0
B-1	3401204973	3290754940	3.35	-	-	-
B-2	1015712460	1015153860	0.05	-	-	-
B-3	157005237	156631070	0.22	-	-	-
B-4	4677989734	4677767120	0.00	-	-	-
B-5	923255957	922858550	0.04	-	-	-
B-6	9525859674	9525841820	0.00	-	-	-
B-7	14835997267	14833996360	0.01	-	-	-
B-8	1214522871	1214153440	0.03	-	-	-
B-9	15886044480	15885369400	0.00	-	-	-
B-10	18049083542	18048006980	0.00	-	-	-

5 LOCAL SEARCH METHODS

In this section, the methods developed to solve the Machine Reassignment Problem will be presented. At first, a randomized local search was developed. This method did not produce good results, but contained some important characteristics such neighborhoods and data structures. In the following, a Simulated Annealing heuristic was proposed. This is the best method presented in this work and uses some important elements from the randomized local search.

5.1 Randomized Local Search

In the first steps of working with the problem, we studies local search algorithms. These methods evolved into a randomized local search and, later, to a Simulated Annealing heuristic. Next, two simple neighborhoods used in these algorithms will be presented. Also, the data structures that permit fast operations will be explained. Finally, the randomized local search algorithm and its results are presented.

5.1.1 Neighborhoods

The first neighborhood, given a valid assignment, moves a process from one machine to another machine. We will call it the **Move Neighborhood**. The size of this neighborhood (number of neighbors of a given a solution) is $O(|\mathcal{P}||\mathcal{M}|)$.

The second neighborhood will be called the **Swap Neighborhood**. Given a valid assignment, a movement of this neighborhood consists in swapping two processes located on two different machines. For example, let $A_1(p_1) = m_1$ and $A_1(p_2) = m_2$. A swap move between processes p_1 and p_2 will produce the following neighbor solution: $A_2(p_1) = m_2$ and $A_2(p_2) = m_1$. The size of this neighborhood is $O(|\mathcal{P}|^2)$. Since it is expected that $|\mathcal{P}| > |\mathcal{M}|$, the size of this neighborhood is likely bigger than the move neighborhood.

5.1.2 Data Structures

In any local search algorithm, there are some core operations that will be used repeatedly: 1) we must verify if a movement of the neighborhood is valid; 2) we must calculate the cost of the neighbor solution; 3) and finally, we must execute the move. Here, the data structures to execute these operations fast and utilization in the context of the move neighborhood will be presented. Its extension to the swap neighborhood is similar.

The used data structures are

- **machineResource**: $|\mathcal{M}| \times |\mathcal{R}|$ integer matrix for storing the usage of each resource in each machine. So, $\mathrm{machineResource}_{mr}$ indicates the usage of resource $r \in \mathcal{R}$ in machine $m \in \mathcal{M}$.
- **transientUsage**: $|\mathcal{M}| \times |\mathcal{R}|$ integer matrix that indicates the usage of transient resources in machines. $\mathrm{transientUsage}_{mr}$ indicates the usage of resource $r \in \mathcal{R}$ in machine $m \in \mathcal{M}$ by processes that were originally in machine m and were moved.
- **serviceMachine**: $|\mathcal{S}| \times |\mathcal{M}|$ boolean matrix used for indicating if a service has any processes in a machine. Therefore, $\mathrm{serviceMachine}_{sm}$ indicates the existence of a process from service $s \in \mathcal{S}$ in machine $m \in \mathcal{M}$. Note that at most one process of a service may be in a machine at some time, so this matrix is composed of booleans.
- **serviceLocation**: $|\mathcal{S}| \times |\mathcal{L}|$ integer matrix where $\mathrm{serviceLocation}_{sl}$ indicates the number of processes of service $s \in \mathcal{S}$ are assigned to machines of location $l \in \mathcal{L}$.
- **serviceNeighborhood**: $|\mathcal{S}| \times |\mathcal{N}|$ integer matrix indicating the number of processes of a given service in a neighborhood. $\mathrm{serviceNeighborhood}_{sn}$ indicates the number of processes of service $s \in \mathcal{S}$ are present in machine of neighborhood $n \in \mathcal{N}$.
- **serviceLocationCount**: $|\mathcal{S}|$ integer vector used to store the current spread factor of the services. $\mathrm{serviceLocationCount}_{s}$ indicates the number of locations that have processes of service $s \in \mathcal{S}$.
- **serviceChanged**: $|\mathcal{S}|$ integer vector. $\mathrm{serviceChanged}_{s}$ indicates the number of processes of service $s \in \mathcal{S}$ that are not assigned to their original machines.
- **serviceChangedCount**: $|\mathcal{P}|$ integer vector where $\mathrm{serviceChangedCount}_{k}$ indicates the number of services that have exactly k processes which changed from their original machines. In fact, its size can be the size of the biggest service.
- **maxServiceChanges**: an integer indicating the maximum number of processes of the same service that changed from machine. It is the service move cost of the solution before multiplying by $\mathrm{weight}_{\mathrm{smc}}$.

Note that this information is redundant. In fact, all this data can be gathered from the current assignment. The reason for keeping all this redundant data updated is to execute the desired operations very fast (mostly in constant time). The memory usage of these data structures is $O(|\mathcal{M}||\mathcal{R}| + |\mathcal{M}||\mathcal{S}| + |\mathcal{S}||\mathcal{L}| + |\mathcal{S}||\mathcal{N}| + |\mathcal{P}|)$. This will usually be dominated by the term $|\mathcal{M}||\mathcal{S}|$. In the challenge ROADEF, there were some big instances but the memory usage never exceeded 64 MB.

Suppose we are moving process $p \in \mathcal{P}$ to machine $m \in \mathcal{M}$. Let $\mathrm{service}(p)$ be the service of process p and $\mathrm{location}(m)$ be the location of machine m. In order to respect the conflict constraint, machine m must not have a process of service s. This is easily achieved by verifying if $\mathrm{serviceMachine}_{\mathrm{service}(p),m} > 0$. For the capacity constraints, we must verify if the requirements of process p plus the current usage of the machine plus the transient usage of the machine are lower than its capacity. Note that transient usage constraints are being verified jointly. However, if $A_0(p) = m$, then the constraints are automatically verified because the requirements of process p will be summed in the usage of the machine but decreased of its transient usage. The cost of the verification is $O(|\mathcal{R}|)$.

Spread constraints are verified in constant time by making sure that the movement will not lower the spread of service s below its minimum requirements. This happens if process p is the only one of $\text{service}(p)$ in its current location and $\text{location}(m)$ already has some processes of $\text{service}(p)$, also $\text{serviceLocationCount}_{\text{service}(p)}$ must be equal $\text{spread}_{\text{service}(p)}$.

The most complicated constraint is the dependency constraint. Let $\text{neighborhood}(m)$ be the neighborhood of machine m. We must guarantee that $\text{neighborhood}(m)$ has processes that satisfy all the dependencies of $\text{service}(p)$. Besides, if process p is the only one in its current neighborhood and there is any process that depend on it, then process p cannot move out. All these verifications can be done in $O(|\mathcal{D}|)$, if we call $\mathcal{D}$ the set of all dependencies. However it is probably less than that since only dependencies involving $\text{service}(p)$ are verified. In summary, the whole verification cost is $O(|\mathcal{R}||\mathcal{D}|)$.

The delta cost of the new solution in relation to the current one can be calculated in $O(|\mathcal{R}||\mathcal{B}|)$. The load cost can be calculated in $O(|\mathcal{R}|)$ by verifying the state of the source and destination machines. Balance cost is calculated in $O(|\mathcal{B}|)$ also by verifying the state of the involved machines if the move is made. Move costs can be calculated in constant time.

Considering the execution of the move, all the data structures must be updated to reflect the new assignment. This can be done in $O(|\mathcal{R}|)$. This cost is due to the update of machineResource and transientUsage matrices, all other data structures are updated in $O(1)$.

5.1.3 Local Search Procedure

Given a current solution, the randomized local search selects one of the k best neighbors using the move neighborhood. If the current solution is a local minimum of this neighborhood, the swap neighborhood is used instead. The search stops when a local minimum of both neighborhoods is found. This process is then repeated from the beginning until the time limit. For our experiments, a value $k = 5$ was used.

This method is a mixture of randomized best improvement local search with a variable neighborhood search. In our experiments, the best improvement heuristic produced better results than the first improvement because it made more significant changes in the solution, even though it took more time. The first improvement heuristic takes the first movement that makes an improvement, however many movements represent an insignificant improvement, especially due to the move costs. The variable neighborhood technique is used to escape a local minimum using a heavier neighborhood. The randomized nature of the algorithm was introduced to utilize all the available time for the challenge, since each execution of the search could potentially lead to a different solution.

Algorithm 1 shows the pseudo-code of the randomized local search. Note that a candidate in a given neighborhood must improve the current solution, otherwise it is not considered a candidate. Also, the choice of a random candidate is made among the best k candidates.

In the reported experiments, the results of a first improvement and best improvement heuristics are presented. A first improvement heuristic is a technique that runs through

Algorithm 1 Randomized Local Search

```
Input: initial solution S
Output: new solution
  while improved current solution do
      if there is a candidate in move neighborhood then
          choose a random candidate
          perform the move
      else if there is a candidate in swap neighborhood then
          choose a random candidate
          perform the swap
      else
          return S
      end if
  end while
```

a neighborhood and chooses the first neighbor of the current solution that improves the cost. The process is repeated until no neighbor improves the solution, which is called a local minimum solution. In our experiments, we used the move neighborhood. In the other hand, the best improvement heuristic runs through a neighborhood and chooses the best neighbor among all of them. The process is also repeated until a local minimum and the move neighborhood was used in the experiments. Usually, the first improvement is a faster method because it does not iterate over all the neighborhood before choosing its target. Also, the best improvement strategy does not guarantee a better solution in the end.

As it turned out in the experiments, a reasonable strategy would be to mix first and best improvement. Choose the first process that generates an improvement, like first improvement. But when considering a process, take the best machine to perform a move, like best improvement. We will call this strategy the fixed-process best improvement (FP).

5.1.4 Results

Table 5.1 presents a comparison between the first improvement and the best improvement heuristics. For each instance, we give the value obtained by the first improvement (FI), the execution time and the relative deviation with relation to the best known values (Table 4.2) calculated as $\mathrm{dev} = (\mathrm{sol} - \mathrm{bkv})/\mathrm{bkv}$. The same information is presented for the Best Improvement heuristic (BI).

It can be seen that many of the executions were stopped because of the time limit of 300 seconds. Therefore, the solutions found are not necessarily local minima. In our case the best improvement strategy showed better results due to its more significant changes in the solution. Note that in the time of testing these algorithms, only instances A were available and a superior algorithm had already been developed when the B instances came out. If choosing a pure local search technique, nor first improvement neither best improvement should be chosen since they could not even achieve a local minimum within the time limit.

Table 5.2 presents the results of the improved local search algorithms. FP shows the results of the fixed-process best improvement heuristic just presented. This heuristic was implemented for comparison purposes after the end of ROADEF challenge due to the

Table 5.1: Comparison between First Improvement and Best Improvement heuristics.

Instance	FI	time [s]	Dev. [%]	BI	time [s]	Dev. [%]
A1-1	44307410	0.003	0.002	44306501	0.003	0.000
A1-2	847270158	1.005	8.969	830092537	0.374	6.760
A1-3	583384093	0.013	0.065	583373292	0.024	0.063
A1-4	340779893	1.940	34.840	305472822	1.113	20.870
A1-5	728088666	0.020	0.070	727578809	0.025	0.000
A2-1	28633567	6.099	15819550	21045707	2.683	11627362
A2-2	1475332347	2.300	85.654	993139356	2.066	24.975
A2-3	1823648909	2.362	41.151	1479599923	1.755	14.522
A2-4	2129868604	8.406	26.741	2014010786	2.067	19.847
A2-5	648316715	2.564	110.793	615442775	1.217	100.104
B-1	4563358943	67.958	34.169	3598178892	51.634	5.791
B-2	1585983541	300.00	56.145	1223973932	124.316	20.504
B-3	3188092285	300.00	1930.911	515164541	300.00	228.175
B-4	7456692502	300.00	59.400	6142786122	300.00	31.313
B-5	9242619109	300.00	901.090	3803071974	300.00	311.920
B-6	10906388931	300.00	14.492	9720586589	300.00	2.044
B-7	37216794965	300.00	150.855	37234576480	300.00	150.975
B-8	11182536419	300.00	820.735	4165958519	300.00	243.012
B-9	21097555215	300.00	32.806	21633896455	300.00	36.182
B-10	41570497323	300.00	130.319	41738070246	300.00	131.248

poor performance of the classical local search strategies in the big instances. RLS states for the results of the randomized local search algorithm. This algorithm always use all the available time, so the column “time” is omitted.

It can be seen that both algorithm outperform the best improvement and first improvement algorithms. Although the fixed-process best improvement heuristic produces worst results in some instances, it outperforms the other heuristics in the big instances due to its fast processing of the neighborhood while still making significant moves. The randomized local search (RLS) was our best algorithm for a while, however it can be seen that its performance is weak in big instances because it relies heavily on the best improvement strategy.

5.2 Simulated Annealing

Simulated Annealing is an approach for combinatorial optimization proposed by Kirkpatrick et al. [1983]. As proposed, moves are chosen randomly with a probability of being accepted that depends on a temperature variable and the quality of the new solution. The temperature is initialized with a high value which has the effect that much worse solutions might be accepted. However, the temperature decreases gradually reducing the probability that worse solutions will be accepted. In the limit, the algorithm behaves like a local search.

Table 5.2: Improved local search algorithms results.

Instance	FP	time [s]	Dev. [%]	RLS	Dev. [%]
A1-1	44307006	0.003	0.001	44306501	0.000
A1-2	861988695	0.04	10.862	783616972	0.782
A1-3	583502292	0.013	0.085	583006016	0.000
A1-4	337809296	0.055	33.665	267766740	5.950
A1-5	727580826	0.015	0.000	727578709	0.000
A2-1	47277617	0.136	26120130	4545638	2511302
A2-2	1440056956	0.09	81.215	985560727	24.022
A2-3	1774226850	0.12	37.326	1396948864	8.124
A2-4	2181492430	0.175	29.813	1782251046	6.056
A2-5	646157082	0.108	110.091	485170386	57.748
B-1	4760518814	0.82	39.966	3490328296	2.620
B-2	1352634450	4.161	33.171	1146912050	12.917
B-3	335672599	7.063	113.834	533086150	239.592
B-4	4678045536	81.44	0.001	6180081570	32.110
B-5	1055122757	16.976	14.283	3838230409	315.728
B-6	9525937752	80.181	0.001	9726660908	2.108
B-7	16647366341	300.0	12.209	37234576780	150.975
B-8	1382321502	30.579	13.816	4258795460	250.656
B-9	15889231476	219.605	0.020	21636685245	36.199
B-10	20158430473	300.0	11.687	41738070446	131.248

For the acceptance probability of a given move, a formula that comes from the Metropolis-Hastings algorithm is usually used. Let $d = v(s) - v(s')$, where $v(s)$ is the objective value of the solution s, s is the current solution and s' is the target solution. If $d \geq 0$, the solution is automatically accepted. Otherwise, the solution is accepted with probability $e^{d/T}$, where e is the Euler constant and T is the current temperature. Note that $\lim_{T \to 0} e^{d/T} = 0$.

The intuition for this technique comes from annealing in metallurgy. In this technique, a material is heated and then it passes by a process of controlled cooling. The slow cooling permits to find configurations of low internal energy, and therefore a material with less defects.

Bertsimas and Tsitsiklis [1993] provide a good mathematical analysis of the behavior of the Simulated Annealing method. Using Markov chains, the convergence of the method can be demonstrated. However, there is no rigorous justification of its speed of convergence. Anyway, this is an heuristic used successfully in practice for various problems.

A detailed empirical study of the technique was performed by Johnson et al. [1989] and Johnson et al. [1991]. The first one studies the behavior of Simulated Annealing for the classical Graph Partitioning Problem. The second paper makes a similar analysis studying Graph Coloring and Number Partitioning.

5.2.1 Algorithm

The proposed Simulated Annealing combines the two neighborhoods presented: move and swap neighborhoods. In each iteration of the method, the move neighborhood is chosen with probability p and the swap neighborhood with probability $1 - p$. Probability $p = 0.7$ was chosen for our experiments.

Given a neighborhood, the Simulated Annealing proceeds by selecting a random neighbor of the current solution. An important detail is the choice of a feasible neighbor of the current solution. Ideally, this would be a random choice among the neighbors. Since the neighborhoods can be large, the feasibility test would incur a large time overhead. We therefore opted for a more efficient sampling strategy.

For the move neighborhood, a process p and a machine k are selected at random. Then, we consider machines $(k + i)\%|\mathcal{M}|$ for $0 \leq i \leq c$, where c is a constant (in the computational experiments we used $c = 100$). The first valid assignment of process p to a machine in the given order, if any, is chosen. Otherwise, process p is considered unmovable and the move is rejected. A similar procedure is used for the swap neighborhood: a process is fixed and a sequence of c other processes is chosen to perform the movement. This strategy guarantees an efficient choice of the neighbor which is constant in the size of the instance.

A cooling cycle of the simulated annealing starts with an initial temperature t_0, holds the temperature constant for n iterations and then reduces it with a cooling rate r. This kind of temperature reduction is called geometrical cooling scheme, since temperatures form a geometrical progression $t_k = t_0 r^k$.

When the current best solution is not updated for $20n$ iterations and the number of accepted moves is less than 0.1% we consider the solution "frozen" (Johnson et al. [1989]). In this case, we reheat the system by increasing the temperature to $t_0/100$. The objective of this reheating procedure is to perform more significant perturbations to the current solution, hoping to escape local minimum.

Algorithm 2 shows an outline of the Simulated Annealing method. It uses a method called sa_move that executes a movement and returns a flag indicating if it was successful. This method is presented in Algorithm 3. In this method, getMoveNeighbor and getSwapNeighbor are methods for choosing a neighbor of a given neighborhood. Also, getCost is a function that returns the difference in cost of the current solution and the solution after performing the indicated move.

5.2.2 Parameter Setting

In the ROADEF challenge, we had available two processors and a time limit of 5 minutes (300 seconds) for solving each instance. We chose to execute two independent threads with different parameters and return the best solution found. So, the choice of parameters of each thread had to be made.

We systematically tested several combinations of parameters values applied to a sub-

Algorithm 2 Simulated Annealing Method

```
Input: initial solution $sol$
Input: initial temperature $t_0$
Input: the allowed time limit TIME_LIMIT
Input: number of iterations per temperature $n$
Input: decrease factor of the temperature $r$
Input: minimum ratio of accepted moves MIN_PERCENT
Input: number of temperature iterations without an update FREEZE_LIM
Output: final solution
    bestSolution $\Leftarrow$ sol
    T $\Leftarrow t_0$
    freeze $\Leftarrow 0$
    while time() < TIME_LIMIT do
        it $\Leftarrow 0$
        ac $\Leftarrow 0$
        for $i = 1 \rightarrow n$ do
            if sa_move(sol, T) then
                $ac \Leftarrow ac + 1$
            end if
            $it \Leftarrow it + 1$
            if sol.cost < bestSolution.cost then
                bestSolution $\Leftarrow$ sol
                freeze $\Leftarrow 0$
            end if
        end for
        T $\Leftarrow$ r $\times$ T
        if $ac/it < MIN\_PERCENT$ then
            freeze $\Leftarrow$ freeze + 1
            if $freeze = FREEZE\_LIM$ then
                freeze $\Leftarrow 0$
                T $\Leftarrow t_0/100$
            end if
        end if
    end while
    return bestSolution
```

Algorithm 3 Simulated Annealing Movement

```
Input: current $sol$
Input: current temperature $T$
Input: probability $p$ of choosing each neighborhood
Output: boolean value indicating if the movement was accepted
    if random() < p then
        move ⇐ getMoveNeighbor(sol)
        delta ⇐ getCost(sol, move)
        if delta < 0 or random() > $e^{-delta/T}$ then
5:          makeMove(sol, move)
            return True
        else
            return False
        end if
10: else
        move ⇐ getSwapNeighbor(sol)
        delta ⇐ getCost(sol, move)
        if delta < 0 or random() > $e^{-delta/T}$ then
            makeSwap(sol, move)
15:         return True
        else
            return False
        end if
    end if
```

set of the instances. The subset chosen was: A1-4, A2-2, A2-3, A2-5, B-1 and B-3, since these were considered the most difficult instances for our method. The values of the parameters we tested were the following: $n \in \{10^4, 10^5, 10^6\}$, $r \in \{0.91, 0.95, 0.97\}$ and $t_0 \in \{10^7, 10^8, 10^9\}$. All the combinations of these values were tested and for each parameter setting and instance, we ran five executions with different seeds. With the values of the average for each considered instance, we calculated scores (relative distance to the optimum) for each parameter setting. Finally, we ranked the results by score. The best score was achieved with parameters: $n = 10^5$, $r = 0.97$ and $t_0 = 10^8$.

This parameter setting performed well in the tested instances, but it could be very slow for some instances, spending too many iterations per temperature combined with a slow decrease of temperature. This could be seen especially for instance B-5, which was not considered in the subset of tested instances, for which some “lighter” parameter settings performed better. To balance this condition, the parameter setting of the second thread was chosen to be faster than the first one (and complementary): $n = 70000$, $r = 0.95$ and $t_0 = 10^8$.

5.2.3 Results

Table 5.3 shows the results of the proposed Simulated Annealing method on the instances A. We present the best results of qualification phase (Qualification), these are the best results over the solutions of all competitors of the challenge, together with its deviation in relation to the best known values of the instances. We also give the results of

Table 5.3: Results of the Simulated Annealing on instances A.

Instance	Qualification	Dev. [%]	SA_v_1	Dev. [%]	SA_v_2	Dev. [%]
A1-1	44306501	0.000	**44306501**	0.000	44306935	0.001
A1-2	777532896	0.000	782071851	0.584	**777533311**	0.000
A1-3	583005717	0.000	**583006016**	0.000	583009439	0.001
A1-4	252728589	0.000	282606396	11.822	**260693258**	3.151
A1-5	727578309	0.000	727578709	0.000	**727578311**	0.000
A2-1	198	9.392	250103	138078	**222**	22.652
A2-2	816523983	2.750	**836004186**	5.202	877905951	10.474
A2-3	1306868761	1.152	**1335318573**	3.354	1380612398	6.860
A2-4	1681353943	0.052	1697598024	1.018	**1680587608**	0.006
A2-5	336170182	9.302	406634034	32.212	**310243809**	0.872

the first version of Simulated Annealing (SA_v_1) and its relative deviation. This was the version used in the qualification phase. It differed from the presented method in the sense that only the move neighborhood was used. And the results of the final version of the Simulated Annealing (SA_v_2) are also presented with their relative deviation.

Note that the solution for some easy instances is worse in the final version. This is because in the qualification version, the Random Local Search Algorithm was executed a couple of times to handle these instances. However, this procedure was not applied in the final version because of the long time this algorithm was taking for big instances (as can be seen in Table 5.2). In general, the results are better compared to the first version of the algorithm.

Table 5.4 shows the results on the instances B. The same information as before is presented, with the exception of column Qualification (and its deviation) because these instances were not used in the qualification phase.

In this table, the superiority of the final version can be perceived. Using only the move neighborhood, the search procedure was trapped frequently in local minima. So, the swap neighborhood helped a lot in exploring other parts of the solution space. Based on these results we conclude that the final Simulated Annealing method is superior than the first method and it is a good option for solving the problem.

Table 5.4: Results of the Simulated Annealing on instances B.

Instance	SA_v_1	Dev. [%]	SA_v_2	Dev. [%]
B-1	3480944379	2.344	**3455971935**	1.610
B-2	1025478846	0.962	**1015763028**	0.005
B-3	793646781	405.577	**215060097**	37.000
B-4	4678188847	0.004	**4677985338**	0.000
B-5	2029086024	119.775	**923299310**	0.005
B-6	9525863457	0.000	**9525861951**	0.000
B-7	15073216111	1.599	**14836763304**	0.005
B-8	4216917662	247.208	**1214563084**	0.003
B-9	15886827597	0.005	**15886083835**	0.000
B-10	18220528250	0.950	**18049089128**	0.000

6 SERVICE ROTATION

6.1 Insight and Modeling

During the development of the work, we noticed an interesting structural characteristic of a solution. This generated an optimization routine that we called **Service Rotation**. This was not incorporated in the final method, but it is worth explaining since it could be a useful element in a solution procedure which can spend more than the alotted 300 seconds.

Let $s \in \mathcal{S}$ be a service. It is known that every process $p \in s$ is assigned to a different machine due to the conflict constraint. We will show that, if keeping every other assignments as they are, any bijection between processes of service s and their respective machines will automatically respect some of the hard constraints.

Since it is a bijection between processes and machines, the conflict constraint will be automatically satisfied: every process will be assigned to a different machine. The spread constraint is also being respected because the set of machines occupied by service s is the same, and therefore the same set of locations. And finally, the dependency constraints are also satisfied. All processes of service s have the same needs in terms of dependencies, and any of them is able to satisfy their dependents. Therefore, we can exchange them between their machines without affecting the dependencies. Having said that, the only hard constraints that are not guaranteed are the capacity constraints (including transient usage).

We will model the problem of finding the best assignment of process to machines as the classical **Assignment Problem**. The problem consists on finding the minimum weighted matching in a weighted bipartite graph. In the classical example, there are n workers and n machines. Each worker performs the job on each machine in a different time. The objective is to find an assignment that minimizes the sum of utilization time of the machines. This is reasonable considering that the cost is propotional to the time machines are on.

In our case, we have a bipartite graph with processes of service s in one part and their respective machines in the other part. There is an edge between process p and machine m if this assignment respects the capacity constraints of machine m. The cost of this assignment, if allowed, can be calculated by the new state of machine m. This cost includes load costs, balance costs and move costs. Figure 6.1 illustrates the bipartite graph that must be created.

Before the procedure, we virtually take out the processes involved of their machines.

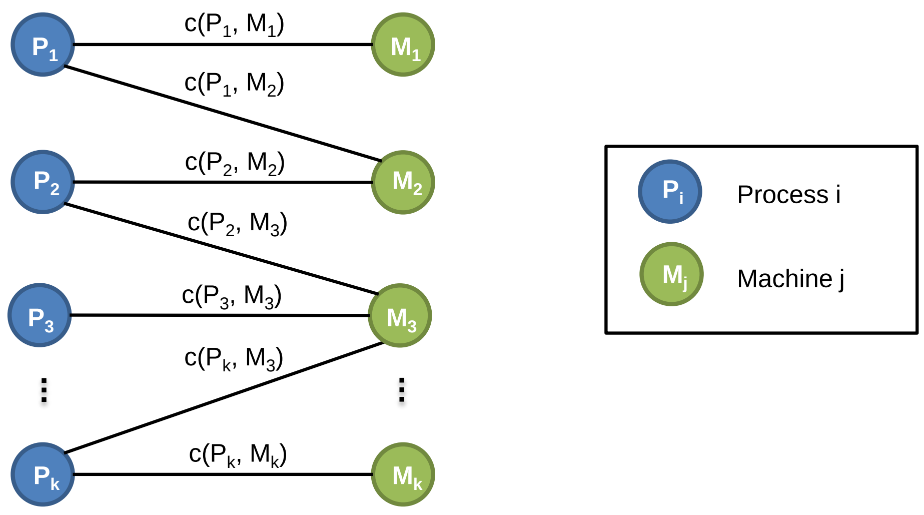

Figure 6.1: Assignment Problem for the service rotation.

Then, for calculating the new state of machine m if process p is assigned to it, we just have to sum its resource requirements. With this information, the capacity constraints may be verified for the existence of the edge. Also, load costs and balance costs are easily calculated.

Process move cost can be included by summing $\mathrm{PMC}(p)$, unless $A_0(p) = m$. The machine move cost is also easily included by summing to the cost of the edge $\mathrm{MMC}(A_0(p), m)$. The only problem with this modeling is the service move cost. It seems not possible to include the service move cost in this context because it is a global cost, while all other costs could be calculated locally.

The most famous algorithm for solving the assignment problem is the Hungarian Algorithm, that runs in $O(n^3)$. The algorithm is also known as the Munkres Assignment Algorithm.

6.2 Results

The service rotation has the drawback of always keeping the processes in the same set of machines. It should, therefore, be combined with some other method for a better exploration of the solution space.

Service rotation was not incorporated in our final method because the strength of our simulated annealing lies in the number of iterations done. Service rotation was more costly than our current neighborhoods. The technique could be also used as a post-optimization method. However, the modeling problem with service move cost resulted in some worse results.

The results of some pre-tests made with instances A are present in Table 6.1. The service rotation can be considered a movement where the neighborhood consists of solution where the processes of some service has been permuted. Then, a movement consists in choosing a service an executing the service rotation optimization with this service. Column SR presents the results obtained with this technique. The initial solution column presents the value of the original assignment for comparison purposes.

Note that some instances are not optimized at all. This is natural for instances where the services do not play an important role. In instance A2-1, for example, all services are composed of a single machine. Looking at the results, we reiterate that this method should be combined with some other optimization procedure. However, we still believe that this might be an interesting approach because it takes advantage of a structural property of the problem.

Table 6.1: Results using Service Rotation.

Instance	Initial Solution	SR	Dev. [%]
A1-1	49528750	44307420	0.002
A1-2	1061649570	1061659570	36.542
A1-3	583662270	583582202	0.099
A1-4	632499600	409192743	61.910
A1-5	782189690	774241960	6.414
A2-1	391189190	391189190	216126524
A2-2	1876768120	1836542091	131.108
A2-3	2272487840	2246721569	73.897
A2-4	3223516130	2306511773	37.253
A2-5	787355300	644687050	109.613

7 TWO-MACHINE NEIGHBORHOOD

From the data presented so far, it can be perceived that instance A2-1 is singular. Table 4.2 shows that its best known value is extremely low and also its lower bound is 0. In fact, this instance has no balance cost and its best solution has a load cost value of 0. Therefore, its cost is composed only of move costs. Also, from most experimental data like that presented in Table 5.2, the gap values for this instance are very large due to its low optimal solution.

While studying this particular instance, we saw that most constraints are trivially satisfied: the instance has no dependencies, no balance costs and all services are composed of one process, so conflict and spread constraints are trivially respected. In some sense, this instance is very similar to a multi-dimensional vector bin-packing problem (see Csirik et al. [1990]) - a multi-dimensional bin-packing where the dimensions are independent.

This encouraged us to develop some methods that could deal with this instance in particular because our methods performed very poorly at the time. The two machines movement is one of these methods, even though the simulated annealing evolved and outperformed it after all. So, it is being presented here as a separate technique which may be interesting for instances with a special structure.

7.1 Algorithm

The idea is to have a costly movement that is capable of performing a better optimization. Given a solution, take two machines and solve the original problem optimally for these machines. It means to solve the problem considering two machines and the processes assigned to them.

Some processes may be fixed because of dependency or spread constraints. For all the other processes, they have two possibilities of assignment. Therefore, a brute-force solution would take $O(2^n)$, where n is the number of free processes. Since each process has two possibilities of assignment, there are 2^n possible assignments. We tried to solve such small subproblems with CPLEX, but the number of performed movements was too low. The solution was to use a simple branch-and-bound solver limiting the number of considered processes. If $n > 20$, choose 20 processes randomly. The achieved solution was not the optimal solution considering the two machines, but the computation was fast enough to perform about ten movements per second.

Although being a brute-force solution, there was some pruning cuts to make the method faster. If the capacity of any of the machines is exceeded at some node of the search tree, the method backtracks. Also, if the cost of any partial solution is more than the best solution found so far, the method prunes the current branch of the tree. Note that some special attention must be taken considering balance cost because the assignment of new processes may lower its cost. So, it is necessary to ignore this cost when computing partial solutions cost. Algorithm 4 shows the pseudo-code of this procedure.

Algorithm 4 Two-machine Neighborhood

```
Input: process $p$
Input: current capacities
Input: current assignment
    if current capacities exceeded then
        return
    end if
    if partial solution cost > best solution then
5:      return
    end if
    if all processes assigned and current solution < best solution then
        update best solution
        return
10: end if
    update capacities of $m_1$
    assign process $p$ to $m_1$
    branch(p+1, capacities, assignment)
    undo updates and assignment
15: update capacities of $m_2$
    assign process $p$ to $m_2$
    branch(p+1, capacities, assignment)
```

7.2 Results

Table 7.1 shows the results of a local search using the two machines movement (RTM). The moves, that consists in choosing two machines, are selected randomly and then the optimization is performed. Although the results are not very good in general, note that the achieved result for instance A2-1 is very acceptable compared to the ones of the qualification method (see Table 5.3). In fact, the load cost was reduced to 0 and the value is composed only of move costs. So, it can be stated that the two machines movement was promising. However, the improvement in our Simulated Annealing heuristic outperformed these results in a simpler way and this idea was not pursued further.

Table 7.1: Results for the Randomized Two Machine movements.

Instance	RTM	Dev. [%]
A1-1	44307208	0.002
A1-2	788728808	1.440
A1-3	583009440	0.001
A1-4	294135970	16.384
A1-5	727593403	0.002
A2-1	996	450.276
A2-2	1186148908	49.263
A2-3	1579477162	22.252
A2-4	1716759534	2.158
A2-5	370576994	20.489
B-1	3785633467	11.303
B-2	1202542636	18.394
B-3	1225478752	680.667
B-4	5261398180	12.471
B-5	4444227935	381.365
B-6	10068911100	5.701
B-7	21722526481	46.418
B-8	5082005877	318.436
B-9	16610387387	4.560
B-10	30315773140	67.963

8 CONCLUSION

The main contribution of this work is a fast heuristic method for solving the Machine Reassignment Problem. The method is a Simulated Annealing meta-heuristic that uses two simple neighborhoods. The main operations are performed very quickly and, therefore, a great part of the solution space can be explored in a short time. Conceptually, the method is very simple, but it is capable of achieving near optimal solutions for very large instances.

We also propose a lower bound for the problem that seems to be good approximation of the optimal solution in most cases. An integer programming formulation has also been presented, even though it seems not practical for solving medium or large instances. At last, two alternative ideas for solving the problem were discussed. The techniques were not used in the final heuristic solution, but may be interesting if we allow more time than 300 seconds to find better solutions.

There were some ideas that we pursued during the development of this work, but did not deserve their own section here. At the beginning, a Constraint Program was formulated, however it was not implemented for comparing with the integer programming formulation. Also, for improving the Simulated Annealing method, we tried to allow visiting infeasible solutions. The challenge in such approach is to develop a mechanism to rapidly guide the search to a feasible solution when the allotted time ends. Some preliminary tests showed that an objetive function penalizing infeasible solutions is not sufficient to achieve this. A path-relinking method was also developed, however most visited solutions were infeasible and the search could not improve the current solutions. Finally, it was tried a lot to develop a constructive method for finding different initial solutions and exploring other areas of the solution space. However it is a very complicated task that deserves its own optimization method for achieving feasibility. Therefore, we decided to use the given solution as a starting point.

The developed method seems to be efficient on the tested instances. For the final phase of the ROADEF challenge, the methods will be tested against other instances. The results of the challenge will be published the 8^{th} June of 2012 and can be verified on the site of the competition (see ROADEF [2011]).

REFERENCES

D. Bertsimas and J. Tsitsiklis. Simulated annealing. *Statistical Science*, 8:10–15, 1993.

János Csirik, J. B. G. Frenk, Martine Labbé, and Shuzhong Zhang. On the multidimensional vector bin packing. *Acta Cybern.*, 9(4):361–369, 1990.

D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by Simulated Annealing. Part I, Graph Partitioning. *Operations Research*, 37:865–892, 1989.

D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 39(3):378–406, 1991.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

ROADEF. Main page for ROADEF challenge 2009. `http://challenge.roadef.org/2009/en/`, 2009.

ROADEF. Main page for ROADEF challenge 2011/2012. `http://challenge.roadef.org/2012/en/`, 2011.

ROADEF/Google. Google ROADEF/EURO challenge 2011–2012: Machine Reassignment. `http://challenge.roadef.org/2012/files/problem_definition_v1.pdf`, 2011. Version 1.