Ann Oper Res
DOI 10.1007/s10479-015-2082-3

Fast machine reassignment

Franck Butelle[1] · Laurent Alfandari[1,2] · Camille Coti[1] · Lucian Finta[1] · Lucas Létocart[1] · Gérard Plateau[1] · Frédéric Roupin[1] · Antoine Rozenknop[1] · Roberto Wolfler Calvo[1]

Abstract This paper proposes a new method for solving the Machine Reassignment Problem in a very short computational time. The problem has been proposed by Google as subject of the Challenge ROADEF/EURO 2012. The Machine Reassignment Problem consists in looking for a reassignment of processes to machines in order to minimize a complex objective function, subject to a rich set of constraints including multidimensional resource, conflict and dependency constraints. In this study, a cooperative search approach is presented for machine reassignment. This approach uses two components: Adaptive Variable Neighbourhood Search and Simulated Annealing based Hyper-Heuristic, running in parallel on two threads and exchanging solutions. Both algorithms employ a rich set of heuristics and a

✉ Franck Butelle
franck.butelle@lipn.univ-paris13.fr

Laurent Alfandari
laurent.alfandari@lipn.univ-paris13.fr; alfandari@essec.edu

Camille Coti
camille.coti@lipn.univ-paris13.fr

Lucian Finta
lucian.finta@lipn.univ-paris13.fr

Lucas Létocart
lucas.letocart@lipn.univ-paris13.fr

Gérard Plateau
gerard.plateau@lipn.univ-paris13.fr

Frédéric Roupin
frederic.roupin@lipn.univ-paris13.fr

Antoine Rozenknop
antoine.rozenknop@lipn.univ-paris13.fr

Roberto Wolfler Calvo
roberto.wolfler@lipn.univ-paris13.fr

1 LIPN, CNRS UMR 7030, Université Paris 13, Sorbonne Paris Cité, 93430 Villetaneuse, France

2 ESSEC Business School, Cergy, France

Published online: 24 December 2015

learning mechanism to select the best neighborhood/move type during the search process. The cooperation mechanism acts as a multiple restart which gets triggered whenever a new better solution is achieved by a thread and then shared with the other thread. Computational results on the Challenge instances as well as instances of a Generalized Assignment-like problem are given to show the relevance of the chosen methods and the high benefits of cooperation.

Keywords Generalized Assignment · Adaptive Variable Neighborhood Search · Simulated Annealing · Hyper-Heuristic · Cooperative Parallel Search

1 Introduction

This paper considers the Machine Reassignment Problem (MRP) which consists in optimizing the usage of available machine resources by reallocating processes to different machines in a cost-efficient way. The reallocation of the processes must satisfy capacity constraints associated with the machines, and other types of constraints linking subsets of processes. This difficult optimization problem was originally proposed by Google for the 2012 ROADEF/EURO Challenge, further denoted by "Challenge".[1]

This problem can be seen as a multi-resource Generalized Assignment Problem (MRGAP) with some additional constraints and a more complex objective function. In the MRGAP, a set of jobs is assigned to a set of machines. Each job has a cost or profit, and should be assigned to a single machine. When assigned to a machine, each job consumes some resource units. Several resources are associated with a machine in the MRGAP, contrary to the simpler Generalized Assignment Problem (GAP) where only one resource per machine is considered, and the capacity or availability of each resource should not be exceeded, for each machine. The aim of the MRGAP is to find a minimum-cost assignment of jobs to machines, each of which are subject to multi-resource capacity constraints. The MRGAP is NP-hard and has practical applications in distributed computer systems. The Challenge problem extends the MRGAP to a more sophisticated objective function mixing several kinds of costs, and to additional constraints on subsets of jobs. Since the problem is close to the MRGAP, we decided to adapt the algorithm proposed in this paper to the MRGAP as well.

The Challenge problem is also connected to another problem called the Vector Bin Packing (VBP) problem, a multidimensional variant of the Bin Packing Problem (BPP). In the simplest form of the BPP, one considers a set of bins of equal capacity and a list of items, each item having a weight, or processing cost, that is supposed not to vary over bins. The objective is to find the minimum number of bins to pack all items. An instance of the VBP problem consists of a set of items with given sizes that can represent services with known demands, and a set of bins that can represent servers, with known capacities. The service demands and the server capacities span across multiple dimensions in the VBP. The objective is to assign each item to one bin in such a way that for each bin, the total size of the items assigned to the bin does not exceed its capacity for every dimension. The VBP is NP-hard, even when restricted to the one-dimensional case (only an asymptotic Polynomial-Time Approximation Scheme exists see e.g. Vazirani 2001).

The aforementioned two problems, MRGAP and VBP, have different objective functions and formulations, but have the same structure of assigning items to agents at minimum

[1] http://challenge.roadef.org/2012.

cost while satisfying multi-dimensional capacity constraints. The MRGAP is closer to the Challenge problem, because they share the same characteristics that the cost function depends on the assignment variables and the resource consumption of an item varies over the agents, which is not the case for the VBP.

Recently there has been renewed interest in the VBP problem and in the MRGAP because they model particularly well the problem of Virtual Machine (VM) placement. Virtualization has been a growing trend in data-centers with the promise of using computational power more efficiently. Many companies have adopted this technology to cut budgets and maintenance costs. However, the performance of this technique depends on the quality of the management layer that schedules or assigns the virtual machines within a pool of machines in the data-center. While doing this assignment, it is important to make sure that no host gets overloaded while minimizing the number of hosts being used. The goal is not only to reduce the upfront investment in hardware, but also to minimize the energy cost of operating the data center, even when hardware may have been over-provisioned. This problem is made difficult by the multidimensional nature of the load. For example, each virtual machine has its own CPU utilization, memory, and disk, network input and output requirements. Likewise, each host has a capacity for each of these dimensions, and the assignment should ensure that the number of hosts is minimized while no capacity constraint is violated. Moreover, these requirements often vary over time, and if one wishes to avoid migration, one can model the problem by having a dimension for each resource, for each time period. As a consequence, the dimensionality of the problem is increased farther. If we assume that when different virtual machines are placed in the same host, their loads across each dimension are summed up, then the problem of assigning virtual machines to hosts is close to the VBP problem and to the MRGAP.

Given the instance sizes and the computational time limit fixed by the Challenge, we have decided to use heuristics, since exact methods are unlikely to finish within the time limit. The proposed method, called Fast Machine Reassignment (FMR), is a Parallel Cooperation Search. This area has receive much attention in the last decade (see for example Bouthillier and Crainic 2005; Crainic and Gendreau 2002; Crainic et al. 2004; James et al. 2009; Ouelhadj and Petrovic 2008; Rattadilok et al. 2005). Our FMR algorithm explores cooperation of an Adaptive Variable Neighborhood Search (AVNS, see e.g. Ropke and Pisinger 2006) and a Simulating Annealing based Hyper-Heuristic denoted by SAHH (for an example of Hyper-Heuristic using Simulated Annealing see Kalender et al. 2013). A *Hyper-Heuristic* is a methodology designed for hard optimization problems that "performs a search over the space of heuristics rather than the space of solutions" (Burke et al. 2013). It selects iteratively some heuristic in a list of heuristics according to some criteria, that can be in some cases the performance of the heuristic over the last iterations, and may accept or not the output solutions during the search as Simulated Annealing does.

AVNS and SAHH are running in parallel on two threads, exchanging solutions from one thread to another with controlled frequency in order to avoid excessive communication between threads. The main novelty in the proposed method is the way we combine these components, which leads to a large number of local search combinations and experimentally gives better results than threads running fully independently. This confirms the result of Lehre and Özcan (2013) according to which the cooperation of heuristics or meta-heuristics may provide much better results than running them independently. However, this paper shows that the performance of mixing move operators relies critically on having the right mixing distribution, which is problem dependent.

In order to end this introduction and before providing the organization of the paper, let us mention some relevant papers dealing with solving the aforementioned problems. For the

GAP and its variants, a survey on the algorithms used to solve them can be found in Cattrysse and Van Wassenhove (1992), Pentico (2007). Some heuristics have been proposed for the MRGAP, see Gavish and Pirkul (1991), Yagiura et al. (2004b). For bin packing problems, there are many books such as Hochbaum (1996), Kellerer et al. (2004) that detail most of the theoretical literature. Many authors have studied the one-dimensional (Maruyama et al. 1977), the two-dimensional (Chung et al. 1982; Lodi et al. 2002; Puchinger and Raidl 2007) and the three-dimensional case (Martello et al. 2000; Miyazawa and Wakabayashi 2007) and have developed heuristics (Spieksma 1994) and exact methods (Caprara and Toth 2001; Han et al. 1994). To the best of our knowledge, for the large dimensional case, the best empirical results are obtained by variants of the most popular heuristic First-Fit Decreasing (FFD). General systems that manage resources in a shared hosting environment can benefit from good heuristics for VBP (see for example Chen et al. 2005), there are far too many of those to be covered extensively here. Focusing on Virtual Machine placement, there are several VM consolidation heuristics currently used in research prototypes and real VM management tools. For example in Wood et al. (2007), a research system that enables live migration of VMs around overloaded hosts uses a heuristic inspired from FFD, taking the product of CPU, memory, and network loads. CPU consumption and loads are also considered in the Challenge problem.

The paper is organized as follows. Section 2 contains the description of the Challenge problem proposed by Google. The mathematical model and its relations with the MRGAP and VBP are given in Sect. 3. We describe our algorithm named FMR[2] (Fast Machine Reassignment), its components and implementation details in Sects. 4 and 5 respectively. We provide then some experimental results in Sect. 6 for both MRP and MRGAP, and finally conclude the paper.

2 Problem description

The aim of the MRP is to improve the usage of a set of machines. A machine has several resource types, like for example RAM and CPU, and runs processes that consume these resources. Initially each process is assigned to a machine. In order to improve machine usage, processes can be moved from one machine to another. Possible moves are limited by hard constraints, such as for example resource capacity constraints, and have a cost. A solution to this problem is a new process-machine assignment which satisfies all the hard constraints and minimizes a given objective cost.

In the following problem description, we keep the notations of the Challenge as much as possible in order to ease readability for researchers who already know the Challenge problem.

2.1 Decision variables

Let $\mathscr{M}$ be the set of machines, and $\mathscr{P}$ the set of processes. A solution is an assignment of each process $p \in \mathscr{P}$ to one and only one machine $m \in \mathscr{M}$; this assignment is noted by the mapping $M(p) = m$. The original assignment of process p is denoted $M_0(p)$. Note that the original assignment is feasible, i.e. all hard constraints are satisfied.

[2] FMR is open source and is distributed under GPL, see http://www.lipn.fr/~butelle/s26.tgz.

2.2 Hard constraints

2.2.1 Capacity constraints

Let $\mathscr{R}$ be the set of resources which is present on each machine, C_{mr} the capacity of resource $r \in \mathscr{R}$ for machine $m \in \mathscr{M}$ and R_{pr} the consumption of resource $r \in \mathscr{R}$ for process $p \in \mathscr{P}$. Then, given an assignment M, the usage U of a machine m for a resource r is defined as:

$$U(m, r) = \sum_{p \in \mathscr{P} \text{ s.t. } M(p)=m} R_{pr}$$

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

$$\forall\, m \in M, r \in \mathscr{R}, \quad U(m, r) \leq C_{mr}$$

2.2.2 Conflict constraints

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

$$\forall\, s \in \mathscr{S}, \quad \forall \{p_i, p_j\} \in s^2,\ p_i \neq p_j \implies M(p_i) \neq M(p_j)$$

2.2.3 Spread constraints

Let $\mathscr{L}$ be the set of locations, a location $l \in \mathscr{L}$ being a set of machines. Note that $\mathscr{L}$ is a partition of the set of machines $\mathscr{M}$. For each $s \in \mathscr{S}$, let $spreadMin_s \in \mathbb{N}$ be the minimum number of distinct locations running at least one process of service s. The constraints are defined by:

$$\forall\, s \in \mathscr{S}, \ \sum_{l \in \mathscr{L}} \min(1, |\{p \in s \mid M(p) \in l\}|) \geq spreadMin_s$$

2.2.4 Dependency constraints

Let $\mathscr{N}$ be the set of neighborhoods, a neighborhood $n \in \mathscr{N}$ being a set of machines. Note that $\mathscr{N}$ is a partition of the set of machines $\mathscr{M}$. If service s_a depends on service s_b, then each process of s_a should run in the neighborhood of a s_b process:

$$\forall\, p_a \in s_a, \exists\, p_b \in s_b \text{ and } n \in \mathscr{N} \text{ such that } M(p_a) \in n \text{ and } M(p_b) \in n$$

Note that dependency constraints are not symmetric.

2.2.5 Transient usage constraints

When a process p is moved from a machine m to another machine m', some resources are consumed twice; for example, disk space is not available on machine m during a copy from machine m to m', and m' should obviously have enough available disk space for the copy. Let $\mathscr{T} \subseteq \mathscr{R}$ be the subset of resources which need transient usage, i.e. require capacity

on both original assignment $M_0(p)$ and current assignment $M(p)$. Then the transient usage constraints are:

$$\forall\, m \in \mathscr{M}, r \in \mathscr{T}, \qquad \sum_{\substack{p \in \mathscr{P} \text{ s.t.} \\ M_0(p)=m \,\vee\, M(p)=m}} R_{pr} \leq C_{mr}$$

Note there is no time dimension in this problem, i.e. all moves are assumed to be done at the exact same time. Then for resources in $\mathscr{T}$ these constraints subsume the capacity constraints.

2.3 Classification of costs in the objective function

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

2.3.1 Load cost

Let SC_{mr} be the safety capacity of a resource $r \in \mathscr{R}$ on a machine $m \in \mathscr{M}$. The load cost is defined per resource and corresponds to the used capacity above the safety capacity. More formally, let us denote the "over safety capacity" by $\delta^1_{mr} = \max\big(0, U(m,r) - SC_{mr}\big)$, then

$$loadCost(r) = \sum_{m \in \mathscr{M}} \delta^1_{mr}$$

A unit cost c^1_r is associated with the quantity $loadCost(r)$.

2.3.2 Balance cost

As having available CPU resource without having available RAM resource is useless for future assignments, one objective of the problem is to balance available resources. The idea is to achieve a given target t_{r_1,r_2} on the available ratio of two different resources r_1 and r_2. Let $\mathscr{B} \subset \mathscr{R}^2$ be the set of pairs of resources (r_1, r_2) which play a role in the expression of the balance cost.

Let us note by $\delta^2_{m,r_1,r_2} = \max\big(0, t_{r_1,r_2} \cdot (C_m mr_1 - U(m,r_1)) - \big(C_{mr_2} - U(m,r_2)\big)\big)$.

The balance cost for (r_1, r_2) is:

$$balCost(r_1, r_2) = \sum_{m \in \mathscr{M}} \delta^2_{m,r_1,r_2}$$

A unit cost $c^2_{r_1,r_2}$ is associated with the quantity $balCost(r_1, r_2)$.

2.3.3 Process move cost

Some processes are painful to move (having a big code and/or using a big amount of data); to model this soft constraint a process move cost is defined. Let c^3_p be the cost of moving the process p from its original machine $M_0(p)$.

$$processMoveCost = \sum_{\substack{p \in P \text{ s.t.} \\ M(p) \neq M_0(p)}} c^3_p$$

2.3.4 Service move cost

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

$$servMoveCost = \max_{s \in \mathscr{S}}(|\{p \in s \mid M(p) \neq M_0(p)\}|)$$

2.3.5 Machine move cost

Let $c^5_{p,m}$ be the cost of moving p from $M_0(p)$ to $M(p) = m$ (if $M(p) = M_0(p)$ then this cost is zero). The machine move cost is then the sum of these costs over all processes:

$$machMoveCost = \sum_{p \in \mathscr{P}} c^5_{p,M(p)}$$

2.3.6 Total objective cost

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

$$\begin{aligned} totalCost = {} & w_1 \sum_{r \in \mathscr{R}} c^1_r \cdot loadCost(r) \\ & + w_2 \sum_{(r_1,r_2) \in \mathscr{B}} c^2_{r_1,r_2} \cdot balCost(r_1, r_2) \\ & + w_3 \cdot processMoveCost \\ & + w_4 \cdot servMoveCost \\ & + w_5 \cdot machMoveCost \end{aligned}$$

In the data provided by the Challenge we have $w_1 = w_2 = 1$.

3 Mixed integer programming formulations

In this section, we give the formulation of the MRP issued from the Challenge, and a formulation of the MRGAP that uses consistent notations with those of the MRP.

3.1 MRP formulation

We give a MIP formulation of the Google machine reassignment problem, where the only binary variables are assignment variables, and all other variables are continuous variables which are used to express some constraints or terms of the objective function. The decision variables are:

- $x_{pm} = 1$ if process $p \in \mathscr{P}$ is assigned to machine $m \in \mathscr{M}$, 0 otherwise
- δ^1_{mr} = number of units of resource r over Safety Capacity on machine m.
- δ^2_{m,r_1,r_2} = number of available units of resource r_2 on machine m which are under the target, expressed with respect to the number of available units of resource r_1 for $(r_1, r_2) \in \mathscr{B}$.
- $y_{ls} = 1$ if at least one process in service $s \in \mathscr{S}$ is assigned to a machine in location $l \in \mathscr{L}$, 0 otherwise (no need actually to set these variables as binary in the model).

– $z =$ maximum number of moved processes over services.

$$\min\ w_1 \sum_{r\in\mathscr{R}} c^1_r \sum_{m\in\mathscr{M}} \delta^1_{mr} \qquad loadCost \quad (1)$$

$$+ w_2 \sum_{(r_1,r_2)\in\mathscr{B}} c^2_{r_1,r_2} \sum_{m\in\mathscr{M}} \delta^2_{m,r_1,r_2} \qquad balanceCost \quad (2)$$

$$+ w^3 \sum_{p\in\mathscr{P}} c^3_p (1 - x_{p,M_0(p)}) \qquad processMoveCost \quad (3)$$

$$+ w^4 z \qquad serviceMoveCost \quad (4)$$

$$+ w^5 \sum_{p\in\mathscr{P}} \sum_{m\in\mathscr{M}} c^5_{pm} x_{pm} \qquad machineMoveCost \quad (5)$$

s.t.

$$\sum_{p\in\mathscr{P}} R_{pr} x_{pm} \le C_{mr} \qquad \forall m \in \mathscr{M}, r \in \mathscr{R} \quad (6)$$

$$\sum_{\substack{p\in P \text{ s.t.}\\ M(p)\neq M_0(p)}} R_{pr} + \sum_{\substack{p\in P \text{ s.t.}\\ M(p)\neq M_0(p)}} R_{pr} x_{pm} \le C_{mr} \qquad \forall m \in \mathscr{M}, r \in \mathscr{T} \quad (7)$$

$$\sum_{p\in s} x_{pm} \le 1 \qquad \forall m \in \mathscr{M}, s \in \mathscr{S} \quad (8)$$

$$\delta^1_{mr} \ge \sum_{p\in\mathscr{P}} R_{pr} x_{pm} - SC_{mr} \qquad \forall m \in \mathscr{M}, r \in \mathscr{R} \quad (9)$$

$$\delta^2_{m,r_1,r_2} \ge t_{r1,r2} \left(C_{m,r_1} - \sum_{p\in\mathscr{P}} R_{p,r_1} x_{pm} \right) - \left(C_{m,r_2} - \sum_{p\in\mathscr{P}} R_{p,r_2} x_{pm} \right) \qquad \forall m \in \mathscr{M}, (r_1, r_2) \in \mathscr{B} \quad (10)$$

$$\sum_{m\in\mathscr{M}} x_{pm} = 1 \qquad \forall p \in \mathscr{P} \quad (11)$$

$$\sum_{l\in\mathscr{L}} y_{sl} \ge spreadMin_s \qquad \forall s \in \mathscr{S} \quad (12)$$

$$y_{sl} \le 1 \qquad \forall s \in \mathscr{S}, l \in \mathscr{L} \quad (13)$$

$$y_{sl} \le \sum_{p\in s} \sum_{m\in l} x_{pm} \qquad \forall s \in \mathscr{S}, l \in \mathscr{L} \quad (14)$$

$$\sum_{p'\in s^b} \sum_{m\in n} x_{p'm} \ge \sum_{m\in n} x_{pm} \qquad \forall (s^a, s^b),\ p \in s^a, n \in \mathscr{N} \quad (15)$$

$$z \ge \sum_{p\in s} \sum_{\substack{m\in\mathscr{M} \text{ s.t.}\\ m\neq M_0(p)}} x_{pm} \qquad \forall s \in \mathscr{S} \quad (16)$$

$$x_{pm} \in \{0, 1\} \quad (17)$$

$$\delta^1_{mr}, \delta^2_{m,r_1,r_2}, y_{ls}, z \ge 0 \quad (18)$$

There are two types of constraints:

(i) *Local constraints* (6–10) that hold for every machine $m \in \mathcal{M}$:
Capacity constraints (6) (see Sect. 2.2.1) express that the total amount of each resource r on a given machine should not exceed the resource capacity. *Transient usage* constraints (7) (see Sect. 2.2.5) state that for a subset of resources $\mathcal{T} \subset \mathcal{R}$, the total resource consumption of processes p that are assigned to machine m or were initially assigned to m, is no more than the capacity. *Conflict* constraints (8) (see Sect. 2.2.2) state that any two processes of the same service s should not be assigned to the same machine. *LoadCost* constraints (9) (see Sect. 2.3.1) define variables δ^1_{mr} as the number of units of resource r over Safety Capacity on machine m, together with non-negativity constraints on these variables. Finally, *BalanceCost* constraints (10) (see Sect. 2.3.2) define variables δ^2_{m,r_1,r_2}, as the number of available units of resource r_2 on machine m under the target, expressed with respect to the number of available units of r_1. These variables will be equal to zero if the target is achieved due to non-negativity of variables, and are used in the objective function to model so-called balance costs.

(ii) *Global constraints* (11–16) that link machines of $\mathcal{M}$ altogether:
Assignment constraints (11) express that each process should be assigned to a single machine.
Spread constraints (12–14) (see Sect. 2.2.3) are separated into three blocks of constraints. Technical constraints (13) and (14) define variables y_{sl} as equal to 0 if no process in service $s \in \mathcal{S}$ is assigned to a machine in location $l \in \mathcal{L}$; otherwise we have $y_{sl} \leq 1$ so in order to contribute to cover the right-hand-side of constraint (12) y_{sl} can be set to one. Constraints (12) state that the number of distinct locations where at least one process of service s should run is at least the threshold $spreadMin_s$.
Dependency constraints (15) (see Sect. 2.2.4) express that if a service p in a service s^a is assigned to a machine in a neighborhood n, then there must be at least one process p' in the service s^b that depends on s^a, that is assigned to a machine in the same neighborhood n. Finally, *serviceMoveCost* constraints (16) (see Sect. 2.3.4) define the service move cost as the maximum number of moved processes over services.

3.2 MRGAP formulation

The MRGAP mathematical formulation can be obtained by relaxing different sets of constraints, since the constraints needed to describe the problem are only: (6), (11) and (17). The objective function changes completely, since the MRGAP objective function takes into account only the process move cost, but in a different way with respect to the MRP. In the MRGAP we have a single cost matrix in the objective function, where c_{pm} is the cost of assigning process p to machine m. Therefore, the MRGAP problem can be formulated as follows:

$$z = \min \sum_{p \in \mathcal{P}} \sum_{m \in \mathcal{M}} c_{pm} x_{pm} \tag{19}$$

s.t.

$$\sum_{p \in \mathcal{P}} R_{pr} x_{pm} \leq C_{mr} \qquad \forall m \in \mathcal{M}, r \in \mathcal{R} \tag{20}$$

$$\sum_{m \in \mathcal{M}} x_{pm} = 1 \qquad \forall p \in \mathcal{P} \tag{21}$$

$$x_{pm} \in \{0, 1\} \tag{22}$$

4 The FMR method

The method proposed in this paper is a Cooperative Parallel Search which runs in parallel two different algorithms on two threads asynchronously. Because of the time limitation and the dual core processor of the Challenge, we have considered a simple cooperative scheme with two threads which communicate their best solution and operate multiple restarts. More sophisticated techniques can be found in Bouthillier and Crainic (2005) (where a pool of solutions is shared between the threads, instead of a single one in FMR, in a solution warehouse). Moreover, in our approach there is no need to have a controller as in Ouelhadj and Petrovic (2008), Rattadilok et al. (2005) that would coordinate solution exchanges between threads.

Our FMR algorithm uses a particular combination of an Adaptive Variable Neighborhood Search (AVNS) and Simulated Annealing based Hyper-Heuristic (SAHH).

The AVNS running on the first thread is based on the idea reported in Ropke and Pisinger (2006) and Pisinger and Ropke (2007) where the probability of choosing among the different neighborhoods is updated based on the best results found so far. Nevertheless, our AVNS algorithm has the particularity to be initialized with a warm start greedy heuristic, which generally improves the initial assignment.

The method running on the second thread is a SAHH. Hyper-heuristics have been defined for the first time in Cowling et al. (2001). The proposed algorithm does not belong to the category of HH which generate heuristics, but to those that only select heuristics. Therefore it can be mapped completely in the classification scheme proposed in Burke et al. (2010) and reported in Burke et al. (2013). The scheme is based on two dimensions: selection of the heuristic search space and move acceptance.

Note that the complete combination of neighborhoods and ways of exploring them gives a potentially very large set of heuristics. Nevertheless, both threads use a learning mechanism for choosing the heuristic or neighborhood to execute (some applications of such learning mechanisms can be found in Pisinger and Ropke 2007; Kalender et al. 2013; Burke et al. 2012).

In this section, for illustration purposes we need to describe some numerical results on some particular instances of the Challenge. A full description of the Challenge instances and numerical experiments will be found in Sect. 6.

4.1 Cooperative Parallel Search

An interesting feature of our parallel cooperation scheme is the fact that the threads are asynchronous and the number of exchanges is controlled to avoid excessive communication between threads. It can been seen as a restart mechanism since each thread uses a new starting solution whenever the other one communicates an improving solution. Another choice we made was to be very modular and use a list of algorithms (AlgoList) to apply for each thread, with a learning mechanism for part of them.

A simplified version of the overall algorithm is described in Algorithm 1. The main aspect of parallelization in this algorithm is that when one thread finds a new `bestKnown` solution, the other thread can replace its current solution (`assign`) by `bestKnown`. In that sense, there is a real cooperation between the two threads. Note that this replacement may occur before the end of the execution of the current algorithm `algo` running on the thread; for easing readability we did not mention this technicality in Algorithm 1.

The cooperation of the two threads is illustrated by Fig. 1 that represents the improvements on the best known solution for each thread running on instance a2_4 (with seed 9). Once

Algorithm 1: Cooperative Parallel Search algorithm

input : M_0: Initial assignment of processes to machines, Problem description, AlgoList1, AlgoList2, TimeOut (or use default values)
output: bestKnown: An assignment of processes to machines & improvement value

```
begin
    bestKnown ← cost of M0 ;
    Create an alarm to stop threads and save bestKnown when TimeOut is reached ;
    Run Threads in parallel
        /*bestKnown is shared between the two threads                    */
        Thread 1: doWork(AlgoList1, M0) ;
        Thread 2: doWork(AlgoList2, M0) ;
    end
end
/**                                                                      */
[h]Each thread is working on its own local copy of the assignment.
procedure
    (
end
doWork(AlgoList, assign)) for i←1 to |AlgoList| do
    algo ←AlgoList [i];
    assign ← algo(M0,bestKnown,assign) ;
    mutual exclusion between threads
        if cost(assign) < cost(bestKnown) then
            bestKnown←assign ;
    end
```

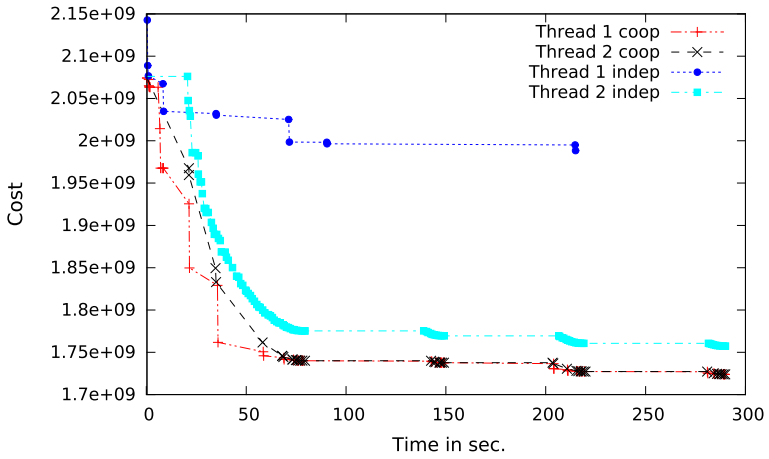

Fig. 1 Comparison of independent threads versus cooperative threads on instance a2_4

in a while, "Thread 1 coop" and "Thread 2 coop" take into account the result of the other thread and introduce improvements and exploration of other neighborhoods. Thread 1 uses a first sequence of heuristics (see Sect. 6.1) followed by AVNS and Thread 2 also uses a first sequence of heuristics followed by SAHH.

We have compared the numerical results of independent threads (taking their best solution only when the time is elapsed) versus our cooperative scheme. Experiments show that the cooperative scheme outperforms the independent solving approach. More details are provided in Sect. 6 with a complete result table.

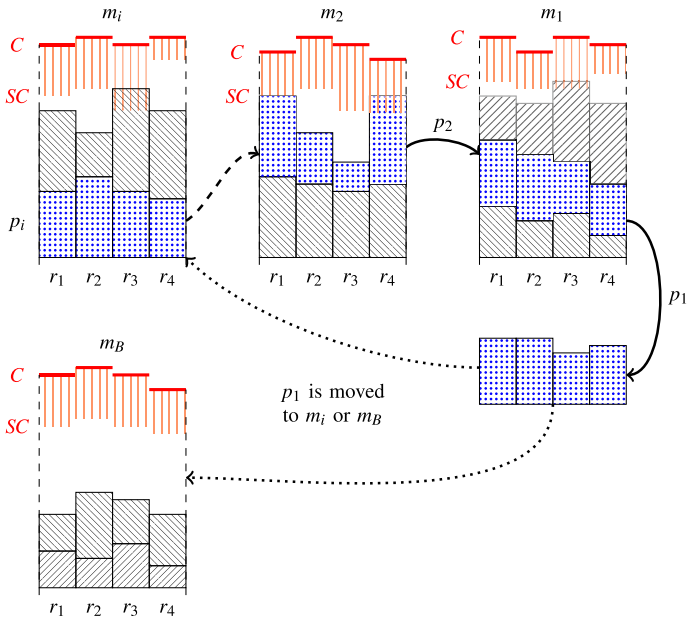

Fig. 2 *Ejection Chain* example (*C* capacity, *SC* safety capacity)

In the following section, we describe the algorithms implemented for AlgoList: Local Search, Greedy, Adaptive Variable Neighborhood Search and Simulated Annealing based Hyper-Heuristic. The specific lists of algorithms chosen for each thread are detailed in Sect. 6.1.

4.2 Local Search and Neighborhoods

Since the problem is quite similar to a GAP (Yagiura et al. 1998), we used the best known moves for the GAP for neighborhood exploration, namely: *Shift*, *Swap* and *Ejection Chain*.

Shift Consider a process p, assigned to a machine m. A *Shift* moves p from m to some other machine m' if no constraint is violated.

Swap Consider two processes p and p' assigned to machines m and m' respectively. A *Swap* exchanges their assignment (if no constraint is violated) i.e. p on m' and p' on m.

Ejection Chain (Yagiura et al. 2004a, 2006) Figure 2 shows the *Ejection Chain* mechanism: choose a process p_1 assigned to a machine m_1 and look for the "best" machine m_B on which p_1 can be assigned. The "best" machine is the one that minimizes the cost when p_1 is moved from m_1 to m_B. Then process p_1 is removed from m_1 and its destination machine will be identified at the end of the *Ejection Chain*. Now find a second process p_2 that can be moved from its machine m_2 to machine m_1. Then find a third process and so on, until a machine m_i gives a process p_i. We stop this procedure at m_i if no more machine has an interesting candidate or when the maximum length of the ejection chain (which is an adjustable parameter), is reached. At the end of this chain process p_1 may be inserted on m_B or on m_i depending on the best move.

Table 1 Cost improvement over computation time ratio for various local search heuristics when using the greedy heuristic

Inst.\algo	Swap-FI	Swap-BI	Shift-FI	Shift-BI
B2	0	0	0	0
B4	5.8×10^4	5.8×10^4	$\mathbf{9.8 \times 10^5}$	9.8×10^5
B6	0	0	0	0
B7	7.3×10^7	1.5×10^6	$\mathbf{1.8 \times 10^8}$	3.4×10^6
B9	1.2×10^3	1.2×10^3	$\mathbf{1.7 \times 10^4}$	1.7×10^4
B10	3.2×10^7	4.4×10^5	$\mathbf{1.1 \times 10^8}$	1.4×10^6

Best ratios are in bold

Local search heuristics According to the policy of acceptance/selection of a new solution (to replace the incumbent) we used the following local search algorithms: first improvement by a shift move denoted by Shift-FI, best improvement by a shift move denoted by Shift-BI, first improvement by a swap move denoted by Swap-FI and Swap-BI the best improvement by a swap move.

For *Ejection Chain*, we use the first improvement acceptance policy only.

4.3 Greedy heuristic

The following greedy heuristic (see Romeijn and Morales 2000 for a similar approach) is performed in order to build an alternate feasible initial solution quite far from the initial assignment M_0 provided with the data set.

First, we partition the services in two classes: the core services are the services submitted to precedence constraints, and the out-of-core services are independent services without such constraints.

Then the processes of the out-of-core services are simply removed from M_0. Thus, we get a partial solution based only on the core services and obviously with less resource consumption than the initial solution.

On this partial (core) solution *Shift* and *Swap* moves are applied to improve the (partial) cost function (*Ejection chain* has not been considered as it typically takes more than 1 min to find an improvement). In practice, we only apply Shift-FI on the core for the following reasons:

- During our experiments, Shift-FI leads to the best ratio of cost improvement over computation time (see Table 1, instances B1, B3, B5 and B8 are not represented because they have *transient usage constraints*). The ratio is 0 for instances B2 and B6 because the initial solution is already a local minimum (for the core) with respect to those two types of move. More details on instances are given in Sect. 6.
- Searching some neighborhoods can be time-consuming and must be avoided (*Shift* is the only one that has linear time complexity according to $|\mathscr{P}|$).
- Combining several neighborhoods on the core provides a low gain on the core after applying the first neighborhood, and may not enable to re-insert all the previously removed processes (too many changes on the core).

In the end, if all the previously removed processes of the out-of-core services are re-inserted, then we obtain a new initial solution for the next step with a lower cost. During the re-insertion step, the algorithm first checks how many machines each process can be assigned to. If there are processes that can be assigned to one machine only (sometimes the original machine) the process is assigned first to prevent infeasibility. Once a process has

been assigned, the data is updated and the algorithm checks again if there exists a process with only one possible insertion point. The procedure goes on until all the remaining processes can be assigned to several machines.

Then for each process p, a function $weight[p]$, summing resource consumptions normalized by the residual capacity of each machine, is computed. Processes are iteratively assigned, in decreasing order of weight (as low-weight processes are easier to place later), to a machine that maximizes cost improvement. The algorithm avoids infeasibility checks whenever it can.

This greedy heuristic is summarized in Algorithm 2. It only runs for instances with no *transient usage constraints* since experiments showed it was generally difficult to reconstruct a feasible solution otherwise.

Algorithm 2: Greedy Heuristic

input : Initial assignment of processes to machines; core ←set of all the processes; out-of-core ← ∅
output: New assignment of processes to machines

if *no transient usage constraints* **then**
  foreach $p \in$ *core* **do**
    $m \leftarrow M_0(p)$;
    if *no precedence relation* **then**
      out-of-core ← out-of-core ∪ $\{p\}$;
      core ← core \ $\{p\}$;
  Apply shift moves to the processes of the core ;
  foreach $p \in$ *out-of-core* **do**
    weight $[p] \leftarrow 0$;
    foreach $m \in \mathcal{M}$ **do**
      foreach $r \in \mathcal{R}$ **do**
        weight $[p]$ ←weight [p] + R_{pr} / C_{mr} ;
  Label all the processes in out-of-core according to an ascending sort of the weights ;
  foreach $p \in$ *out-of-core* **do**
    BestMachine ← ∅ ;
    MaxGain ← 0 ;
    foreach $m \in \mathcal{M}$ **do**
      if *p can be assigned to m* **then**
        gain ← gain associated to assigning p to m ;
        if *gain* > *MaxGain* **then**
          MaxGain ←gain ;
          BestMachine ← m ;
    Assign p to BestMachine ;

4.4 Adaptive Variable Neighborhood Search

The neighborhoods described in Sect. 4.2 are indexed by i below. The AVNS procedure dynamically changes the current neighborhood (Ropke and Pisinger 2006; Pisinger and Ropke 2007). A learning mechanism is used for choosing the next neighborhood. It is based on a scoring function of the neighborhood. More precisely, we use a roulette wheel selection where the score $score[i]$ of neighborhood i equals one initially, and then is updated as $score[i] \leftarrow (1-r)score[i] + r\frac{p[i]}{\theta[i]}$, where $p[i]$ is the number of times i has improved the solution, $\theta[i]$ counts the number of times i has been used, and $r \in [0, 1]$ is a tuning parameter.

According to the size and/or the structure of the instance (e.g. $|\mathscr{M}| \geq 10{,}000$ and/or $|\mathscr{P}| \geq 100{,}000$), we may decide before starting the exploration of the neighborhood by the AVNS algorithm, that some types of moves (typically the *Ejection Chain*) have to be excluded from the set of possible moves.

The AVNS procedure is described in Algorithm 3.

Algorithm 3: Adaptive Variable Neighborhood Search

input : Initial assignment of processes to machines; NH: set of neighborhoods (Local Search)
output: New assignment of processes to machines

foreach $i \in NH$ **do**
  $score[i] \leftarrow 1$; $p[i] \leftarrow 1$; $\theta[i] \leftarrow 1$;
while *time remains* **do**
  foreach $i \in NH$ **do**
    $score[i] \leftarrow score[i](1-r) + r\frac{p[i]}{\theta[i]}$;
  current $\leftarrow 0$;
  foreach $i \in NH$ **do**
    interval $[i] \leftarrow [\text{current}, \text{current} + score[i]]$;
    current $\leftarrow$ current $+ w[i]$;
  MaxFitness $\leftarrow$ current ;
  Choose randomly rand $\in [0, \text{MaxFitness}]$;
  Select $i \in$ NH **s.t.** rand $\in$ interval $[i]$ and i is not the previous neighborhood used
  $\theta[i]++$;
  Apply neighborhood i ;
  gain $\leftarrow$gain associated to applying neighborhood i ;
  if $gain > 0$ **then**
    $p[i]++$;

4.5 Simulated Annealing based Hyper-Heuristic (SAHH)

The proposed selection Hyper-Heuristic framework, called Simulated Annealing based Hyper-Heuristic (SAHH), alternates between two Hyper-Heuristic Selection Strategies (HHSS). A HHSS is defined as a combination of a heuristic selection method and an acceptance method. Indeed, as shown in Bilgin et al. (2006) the computational results might be improved by combining different heuristic selection methods with different acceptance methods. The proposed SAHH framework moves from one HHSS to the other based on the following simple greedy criteria: if a HHSS does not improve the best solution found so far during t_{frozen} seconds, it ends and the other one starts. Following the classification of Burke et al. (2013), the first HHSS (hereafter referred as *Temperature Descent*) uses a *simple random* method to select a heuristic among a set of available heuristics, and accepts new solutions according to the Simulated Annealing function (Kirkpatrick et al. 1983), that allow to *accept some non-improving* solutions with probabilities depending on their scores. Variants of Simulated Annealing are useful as move acceptance components in hyper-heuristics as shown in Bai and Kendall (2005), Dowsland et al. (2007), Bai et al. (2012).

In what follows, we use SR for Simple Random, and SA for Simulated Annealing with reheating, based on the definition in Burke et al. (2012).

The second HHSS (*FastMinimum*) performs the selection according to a *history-based choice function* and always *rejects non-improving* solutions. The proposed framework is related to the one described in Kalender et al. (2013), which can be seen as a mix of

those two strategies, where a *history-based* choice function selects the heuristic and where *non-improving solutions can be accepted* with time-decreasing probabilities. In the next paragraphs we detail the HHSS used by SAHH.

InitialTemperature() The HHSS called Temperature Descent makes use of a *temperature* to compute the probability of accepting a non-improving move. This temperature decreases with time, following a predefined function, but its initial value comes as a parameter to the algorithm and should be fitted to the instance: if it is two low, the algorithm will reject all non improving solutions; two high and it will spend most of its time wandering regardless of the solution costs. We used the following heuristic to compute the initial temperature: during the first time interval t_{init}, we randomly try at most $\mathcal{N}_{feas}$ feasible moves (*Shift* or *Swap*) starting from the initial solution and we store the absolute value of the gain for each move. We then choose the initial temperature T_0 as the median of these stored values, which approximately leads to an initial acceptance rate of 50 % of non-improving moves in *TemperatureDescent()*.

TemperatureDescent (T_0) This HHSS iteratively calculates a new temperature T as a function $f_T(T_0, t_r, t_{descent})$, of the initial temperature T_0, the remaining computational time for *this* descent t_r and the time allocated to the descent $t_{descent}$. Experimentally the following function

$$\left(\frac{t_r}{t_{descent}}\right)^2 T_0$$

has been used.

Then a move is chosen randomly among the implemented ones (i.e. *Shift* and *Swap*), leading to a candidate configuration R_{new}. If no feasible move has been identified during a time period t_{lookup}, then this function stops. Otherwise, it decides whether it accepts the new configuration R_{new} or stays on the previous one R_{old}: R_{new} is accepted if its cost is lower, or else with a probability $p = \exp\left(\frac{\text{cost}(R_{\text{old}}) - \text{cost}(R_{\text{new}})}{T}\right)$ (where T is the temperature computed before). If no move has been accepted during t_{frozen} seconds, then this HHSS stops and the second one executes. Otherwise, it lasts at most $t_{descent}$ seconds.

Figures 3 and 4 show two runs of *TemperatureDescent()* repeated 10 times, respectively on data sets B4 and B10 of the Challenge (see parameter settings in Sect. 6.1). '+' points represent the temperature (left Y-axis), 'x' points represent the cost of the current configuration (right Y-axis). Note that on data set B4, the value of the current solution quickly falls in local minimum and that whenever the temperature starts over from T_0, it gives the function a chance to look at other regions of the search space. Nevertheless, when applied on instances such as data set B10, the SA move acceptance criteria slows down the decrease of the cost.

Therefore the following second HHSS is designed to cope with such instances.

FastMinimum() The learning mechanism for selecting a move is designed as follows. The HHSS keeps an average ratio of *cost gain* over *time spent* for past ten moves, according to the category of moves (*Shift* or *Swap*). The category with the best ratio is first chosen for the next move. Then, the candidate move is selected randomly in this category. If it improves the current solution, it gets accepted; otherwise, it is rejected and in any case the ratio is updated. If the current solution has not been improved during a time interval t_{frozen}, the HHSS stops and the TemperatureDescent HHSS starts. Otherwise it goes on until the global computational time is elapsed.

The SAHH overall framework is described by the pseudo-code given in Algorithm 4.

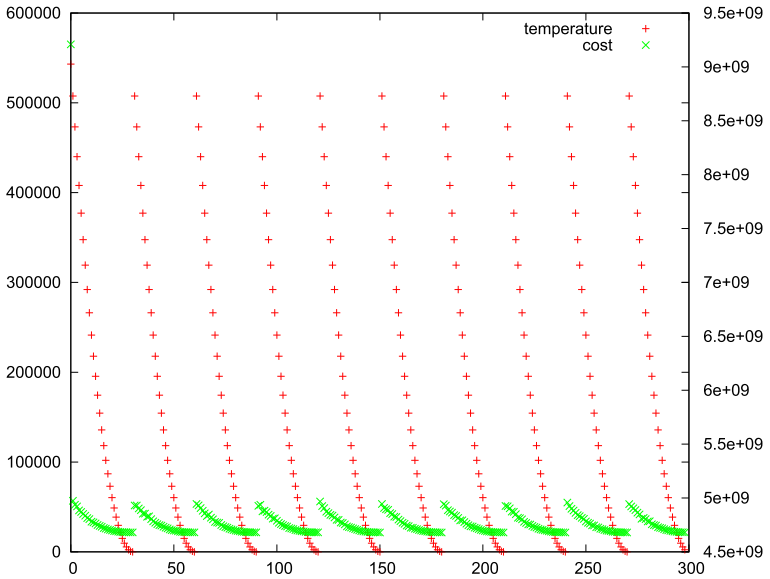

Fig. 3 *TemperatureDescent()* iterated on B4

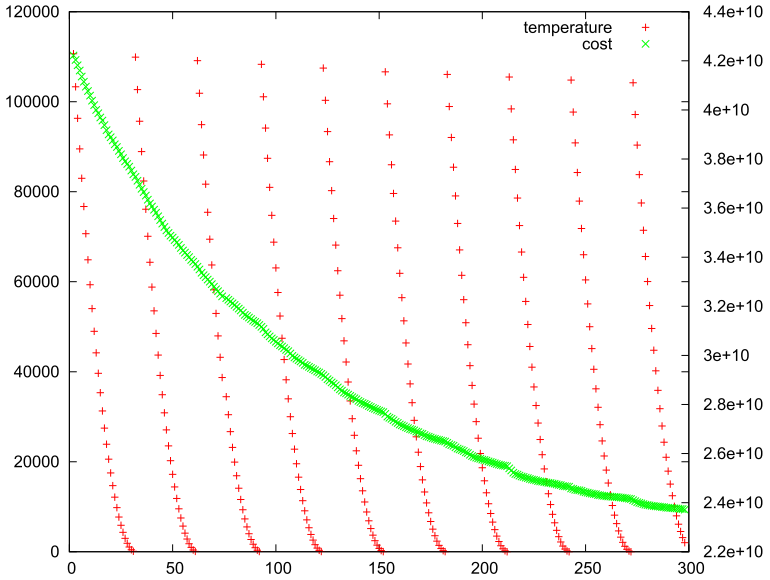

Fig. 4 *TemperatureDescent()* iterated on B10

5 Implementation details

5.1 Communication restriction

Two threads are used to run on the two cores of the reference computer. In order to achieve maximum benefit of these two threads, the code was designed with the aim of limiting communication and synchronizations between the threads.

Algorithm 4: SAHH algorithm

```
T_0 ← InitialTemperature();
while time remains do
    (BestCost,Sol) ←current best solution over both threads;
    FastMinimum(Sol) [1] ;          // during t_descent Sect. or if no improvement during t_frozen,
    if BestCost has not been improved then
        TemperatureDescent(T_0, Sol) [1] ; // during t_descent or if no improvement during t_frozen,
```

[1] FastMinimum and TemperatureDescent start from Sol

Only one shared variable must be accessed in mutual exclusion: the one which is used to store the current best solution found by the threads. This variable is updated most often by the SAHH executed on the second thread. To avoid excessive communication (that is time consuming), in SAHH the frequency of updates is controlled by excluding exchanges during at least t_{frozen} seconds.

The first thread updates this variable on major improvements only, and when the remaining delay is becoming too short.

All the rest of the data is private to each thread and bound to it, or accessed in read-only mode during the parallel execution.

5.2 Compact implementation of a partition of an integer set

For the purpose of the Challenge, the specification of the underlying hardware system states that available memory is limited to 4GB. To this end, all data structures are designed to be compact and to allow fast access during execution of the optimization algorithm.

Intermediate solutions are stored during the search. A solution is specified by a partition of the set of processes into subsets of processes on each machine. A naive implementation of this partition of processes may be highly memory consuming. For example, using an assignment matrix of size $|\mathscr{M}| \times |\mathscr{P}|$ would use a lot of memory and induce a high running time to extract the subset of processes assigned to some machine for large values of $|\mathscr{P}|$.

A second naive implementation could use a linked list for each subset of processes that are assigned to a given machine. Linked list implementation is more compact than the assignment matrix since it uses only $|\mathscr{P}|$ nodes in total. However, such an implementation would slow down the processor because of a very high cache miss rate (since nodes are not allocated contiguously in the memory).

Moreover, creating a new solution starting from some current solution involves allocating and then copying the current solution. This duplication will be highly time-consuming for both aforementioned naive implementations.

Our implementation of the partition of processes is also based on a linked list to store the subset of processes assigned to a given machine. But in FMR, all these lists are stored in one vector of size $|\mathscr{P}|$. On top of that, we use a second vector of size $|\mathscr{M}|$ to keep track of the entry point of each linked list.

In Fig. 5 we show a small example with 4 machines and 6 processes. In this example, processes $\{5, 0, 2\}$ are assigned to machine 0, no processes are assigned to machine 1, processes $\{1, 3\}$ are assigned to machine 2 and process 4 is assigned to machine 3. Notation -1 represents the end of the list.

Our linked list is more compact than classical linked list implementations: we use vectors of short int (size 2 Bytes) instead of pointers (size 8 bytes on a 64 bits system). When our data structure that partitions processes needs to be duplicated, less time is needed for memory

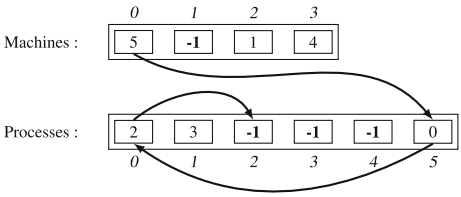

Fig. 5 Example of a compact implementation of a partition of a set

allocation and copy. As memory allocation is contiguous (there are 2 vectors), the cache hit rate is highly improved: when the data structure is used at run-time, we achieve better spatial locality in the cache.

6 Numerical results

6.1 Parameter settings

The different values of the parameters used in our FMR algorithm are the following ones:

- Parameter for the roulette wheel selection of the AVNS (Sect. 4.4): $r = 0.2$
- Parameters for the simulated annealing (Sect. 4.5):
  - Time for finding the initial temperature: $t_{init} = 30$ s
  - Maximal number of moves to find the initial temperature: $\mathcal{N}_{feas} = 10{,}000$
  - Time for each temperature descent: $t_{descent} = 60$ s
  - Time for finding a feasible move: $t_{lookup} = 5$ s
  - Time for accepting a move: $t_{frozen} = 10$ s

All the experiments were run on a computer with an Intel Core i7-2600 CPU at 3.40 GHz. We have arbitrarily chosen the default random seed to be 16. The list of algorithms on Thread 1 AlgoList1 is (Greedy, Shift, EjectionChain, AVNS) and AlgoList2 is (ExtendedBestShift, EjectionChain, SAHH) for Thread 2.

6.2 Problem sizes for MRP

In the Challenge, set sizes are limited to the following maximum values:

- Number of machines $|\mathcal{M}| = 5000$
- Number of resources $|\mathcal{R}| = 20$
- Number of processes $|\mathcal{P}| = 50{,}000$
- Number of services (of cardinality >1) $|\mathcal{S}'| = 5000$ $(\mathcal{S}' \subset \mathcal{S})$.
- Number of neighborhoods $|\mathcal{N}| = 1000$
- Number of dependencies $|\mathcal{D}| = 5000$
- Number of locations $|\mathcal{L}| = 1000$
- Number of balance costs $|\mathcal{B}| = 10$

All other integers are indices or 32-bits unsigned integers. As usual in the ROADEF/EURO Challenge, three data sets have been provided:

- Data set A: $|\mathcal{P}|$ is limited to 1000. This small data set is public and is used during the qualification phase;
- Data set B: $|\mathcal{P}|$ varies from 5000 to 50,000. This medium/large data set is public and is used to evaluate proposed solvers;

Table 2 Instances B

Instance	B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	
$\lvert\mathscr{P}\rvert$	5000	5000	20,000	20,000	40,000	40,000	40,000	50,000	50,000	50,000	
$\lvert\mathscr{M}\rvert$	100	100	100	500	100	200	4000	100	1000	5000	
$\lvert\mathscr{R}\rvert$	12	12	6	6	6	6	6	3	3	3	
$\lvert\mathscr{T}\rvert$	4	0	2	0	2	0	0	1	0	0	
$\lvert\mathscr{L}\rvert$	10	10	10	50	10	50	50	10	100	100	
$\lvert\mathscr{S}\rvert$	2512	2462	15,025	1732	35,082	14,680	15,050	45,030	4609	4896	
#DS	4412	3617	16,560	40,485	14,515	42,081	43,873	15,145	43,437	47,260	
Scores %											Total
First	0.41	0.00	0.01	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.43
Our	7.32	0.04	0.08	0.00	0.00	0.00	0.12	0.21	0.06	0.01	7.85
Last	2.27	8.57	6.92	14.04	7.85	5.33	41.04	6.60	23.24	43.06	158.92

Table 3 Instances X

Instance	X1	X2	X3	X4	X5	X6	X7	X8	X9	X10	
$\lvert\mathscr{P}\rvert$	5000	5000	20,000	20,000	40,000	40,000	40,000	50,000	50,000	50,000	
$\lvert\mathscr{M}\rvert$	100	100	100	500	100	200	4000	100	1000	5000	
$\lvert\mathscr{R}\rvert$	12	12	6	6	6	6	6	3	3	3	
$\lvert\mathscr{T}\rvert$	4	0	2	0	2	0	0	1	0	0	
$\lvert\mathscr{L}\rvert$	10	10	10	50	10	50	50	10	100	100	
$\lvert\mathscr{S}\rvert$	2529	2484	14,928	1190	34,872	14,504	15,273	44,950	4871	4615	
#DS	4164	3742	15,201	38,121	20,560	39,890	43,726	12,150	45,457	47,768	
Scores %											Total
First	0.00	0.02	0.00	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.03
Our	4.65	0.25	0.08	0.00	0.00	0.00	0.29	0.00	0.02	0.01	5.31
Last	1.84	10.53	4.86	15.78	10.58	5.44	43.71	21.77	21.17	41.61	177.28

Table 4 Comparison between independent versus cooperative threads

Inst.	Indep #best (/20)	Coop #best (/20)	Mean indep %gap to best	Mean coop %gap to best
B1	4	2	14.735	14.721
B2	10	10	0.116	0.113
B3	13	6	4.040	4.366
B4	2	17	0.001	0.001
B5	3	16	0.462	0.027
B6	11	9	0.000	0.000
B7	0	2	0.092	0.092
B8	0	16	2.367	2.365
B9	6	9	0.075	0.074
B10	7	13	0.001	0.001
X1	2	0	11.056	11.056
X2	4	16	1.386	1.308
X3	11	8	1526.252	1492.612
X4	2	16	0.001	0.001
X5	6	14	334.953	211.005
X6	6	12	0.000	0.000
X7	0	20	0.255	0.254
X8	8	12	186.258	178.847
X9	9	9	0.008	0.008
X10	10	9	0.000	0.000
Sum	114	216		

– Data set X: $|\mathscr{P}|$ varies from 5000 to 50,000. This medium/large data set is private and is used to evaluate proposed solvers.

The score of a team for an instance is computed as the sum of normalized differences between the final objective function obtained and the best among all participants.[3]

6.3 Numerical results on instances B et X for MRP

Tables 2 and 3 show details on instances used for the Challenge. For instances B and X, the number of neighborhoods is always equal to 5 and the number of services containing at least two processes ($|\mathscr{S}'|$) is 1000, except for B1, B2 and X1 and X2 (which are at 500). #DS stands for the total number of dependencies among services. "First" stands for the results of the best team. "Last" stands for the results of the last team among all those that were able to complete the Challenge.

Note that our score is very close to the results of the best team except for two instances: B1 and X1.

We also tried to increase the computing time to 30 min (with a maximum length of ejection chain extended to fifty), and we then improved four of the best known results.

[3] For more detailed results and information see http://challenge.roadef.org/2012/en/results.php.

Table 5 MRGAP instances C

$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{M}\rvert$	$\lvert\mathscr{R}\rvert$	lb	TS-CS	FMR	TS-WCSP	CPLEX
100	5	1	1931	1931	1970	1933	1931
100	5	2	1933	1933	1995	1933	1933
100	5	4	1943	1943	1953	1944	1943
100	5	8	1950	1950	1989	1956	1950
100	10	1	1402	1402	1478	1402	1402
100	10	2	1409	1409	1425	1411	1409
100	10	4	1419	1419	1464	1419	1419
100	10	8	1435	1436	1503	1435	1435
100	20	1	1243	1245	1243	1245	1243
100	20	2	1250	1251	1252	1253	1250
100	20	4	1254	1257	1255	1258	1254
100	20	8	1267	1269	1268	1267	1272
200	5	1	3456	3456	3492	3460	3456
200	5	2	3461	3461	3500	3462	3461
200	5	4	3466	3466	3504	3469	3466
200	5	8	3473	3473	3532	3478	3474
200	10	1	2806	2807	2923	2811	2806
200	10	2	2811	2812	2866	2812	2812
200	10	4	2819	2821	2935	2823	2819
200	10	8	2833	2837	2929	2842	2842
200	20	1	2391	2393	2412	2394	2391
200	20	2	2397	2398	2412	2403	2398
200	20	4	2408	2409	2422	2415	2415
200	20	8	2415	2422	2428	2423	2419

For the Challenge, 82 teams registered and the organizers decided to set the qualification threshold to the best 30 teams selected on A instances. Our team was ranked 14 among 30 in the qualifying stage. The final ranking was computed on a score based on instances B and X and our team was ranked among the top 20 teams. Note that the gap between our results and the best team is mainly due to some under-performance for one of the 10 instances, both for B and X instances (B1, X1).

For 18 out of the 20 instances, the difference is very small: the total gap of 5.31 for X is mainly due to one gap of 4.65 for instance X1 (average gap = 0.07 for the 9 other X instances vs. 0.003 for the Best average over 10), and the total gap of 7.85 for B is mainly due to one gap of 7.32 for instance B1 (average gap = 0.06 for the 9 other X instances vs. 0.04 for the best average over 10).

When instances become bigger and more complex to solve, our results become more competitive. Our approach seems to be robust on the variability of the input instances since we did not tune our code on the Challenge instances. This allows our approach to be effective also on MRGAP as shown in Sect. 6.5. Before reporting results on the MRGAP, we conclude on the Challenge problem by comparing our cooperative approach to running AVNS and SAHH independently on the two threads without sharing solutions.

Table 6 MRGAP instances D

$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{M}\rvert$	$\lvert\mathscr{R}\rvert$	lb	TS-CS	FMR	TS-WCSP	CPLEX
100	5	1	6353	6357	6620	6370	6358
100	5	2	6352	6359	6471	6380	6360
100	5	4	6362	6379	6524	6404	6386
100	5	8	6388	6425	6613	6500	6428
100	10	1	6342	6361	6415	6418	6381
100	10	2	6340	6378	6453	6411	6419
100	10	4	6361	6430	6476	6516	6468
100	10	8	6388	6478	6533	6679	6528
100	20	1	6177	6231	6289	6305	6280
100	20	2	6165	6261	6302	6389	6316
100	20	4	6182	6321	6339	6529	6406
100	20	8	6206	6482	6440	6736	6588
200	5	1	12,741	12,751	12,951	12,760	12,750
200	5	2	12,751	12,766	13,061	12,778	12,766
200	5	4	12,745	12,775	13,027	12,799	12,762
200	5	8	12,755	12,805	12,862	12,844	12,787
200	10	1	12,426	12,463	12,592	12,478	12,457
200	10	2	12,431	12,477	12,614	12,533	12,482
200	10	4	12,432	12,496	12,640	12,586	12,532
200	10	8	12,448	12,571	12,667	12,812	12,577
200	20	1	12,230	12,312	12,466	12,409	12,393
200	20	2	12,227	12,332	12,491	12,442	12,425
200	20	4	12,237	12,396	12,539	12,605	12,472
200	20	8	12,254	12,485	12,578	12,918	12,548

6.4 Comparison of independent versus cooperative scheme

We compared experimentally the results of independent threads, taking the best solution of AVNS and SAHH only when the time is elapsed, versus our cooperative scheme. Each instance B and X have been run 20 times with different seeds.

The result is given in Table 4. In some cases the two approaches give the same result, which explains why the sum of the number of times the independent scheme is better ("Indep #best") and the number of times the cooperative scheme is better ("Coop #best") is not equal to 20. The "Mean indep (or coop) %gap to best" is computed as follows: for each of the 20 runs, calculate the difference between the independent (resp. cooperative) solution value and the Challenge best known value, divided by this best known value, and compute the mean of these 20 ratios.

We can see that the cooperative scheme globally outperforms the independent one. More precisely, the cooperative scheme finds a strictly better solution for 54 % of the cases (216 over 400 runs of the B and X instances) versus 28.5 % for the independent scheme. Also, the mean gap to the best known value is strictly better for the cooperative scheme for 10 of the 20 instances B and X, and equal for 9 of them. It is significantly better for 4 instances, whereas

Table 7 MRGAP instances E

$\lvert\mathscr{P}\rvert$	$\lvert\mathscr{M}\rvert$	$\lvert\mathscr{R}\rvert$	lb	TS-CS	FMR	TS-WCSP	CPLEX
100	5	1	12,681	12,681	12,716	12,753	12,681
100	5	2	12,692	12,692	12,756	12,727	12,692
100	5	4	12,810	12,812	12,934	12,893	12,810
100	5	8	12,738	12,738	12,765	12,876	12,749
100	10	1	11,577	11,577	11,656	11,712	11,584
100	10	2	11,582	11,587	11,675	11,665	11,612
100	10	4	11,636	11,676	11,759	11,864	11,753
100	10	8	11,619	11,701	11,765	11,836	11,739
100	20	1	8436	8447	8543	8655	8565
100	20	2	10,123	10,150	10,298	10,471	10,251
100	20	4	10,794	11,029	11,135	11,271	11,443
100	20	8	11,224	11,610	11,847	11,957	12,458
200	5	1	24,930	24,933	24,966	25,002	24,930
200	5	2	24,933	24,936	25,048	25,024	24,933
200	5	4	24,990	24,999	25,110	25,091	25,003
200	5	8	24,943	24,950	25,061	25,090	24,943
200	10	1	23,307	23,312	23,351	23,414	23,321
200	10	2	23,310	23,317	23,391	23,538	23,325
200	10	4	23,344	23,363	23,412	23,628	23,543
200	10	8	23,339	23,412	23,554	23,714	23,744
200	20	1	22,379	22,386	22,510	22,815	22,457
200	20	2	22,387	22408	22,477	22,834	22,558
200	20	4	22,395	22,439	22,574	22,990	22,782
200	20	8	22,476	22,614	22,931	23,057	23,482

the independent scheme is strictly better for only one instance (B3). As mentionned in the introduction, these results confirm the benefits of cooperation on this particular problem.

6.5 Numerical results on the MRGAP

We have chosen to compare our code to the one of Yagiura et al. (2004b), one of the best known algorithms for the MRGAP problem, even if FMR was not specifically designed for this problem. So we adapted our code at a minimum for the MRGAP problem and tried it over C, D and E instances.[4] We would like to thank Prof Yagiura who kindly supplied us with initial solutions as well priv. comm.. These initial solutions were output by their simpler and modified version of their algorithm "TS-CS", named "TS-noCS" (that can be found in Yagiura et al. 2004b).

In the results presented in Tables 5, 6 and 7, TS-CS stands for Tabu Search with Chained Shift neighborhood. TS-WCSP stands for a general solver for the Weighted Constraint Satisfaction Problem (see Nonobe and Ibaraki 2001). The results show that globally our code seems to be competitive with TS-WCSP and CPLEX as soon as instances become harder.

[4] See http://www-or.amp.i.kyoto-u.ac.jp/~yagiura/mrgap.

Since our code is really not specific for MRGAP, we think it is able to be a good start for other Generalized Assignment-like problems.

7 Conclusion

We have presented in this paper our contribution to the Challenge Roadef 2012 on Machine Reassignment. Our FMR method provides solution values that are very close to those of the Challenge winner on almost all instances. Moreover, the same code with just a few modifications performs well on a high proportion of instances of the MRGAP problem, although it was not designed for this problem. Experimental results show that FMR is quite robust and behaves better when instances become harder. Let us point out that, in spite of strong operational Challenge constraints (time limitation, specific computer) having a real impact on our algorithm, several ideas developed here can actually be used in a more general context. In particular, the heuristics cooperation and communications issues are core to our approach.

As noticed before, the time limitation given by the Challenge prevents us from using directly mathematical programming here. Nevertheless, the latter approach could be used to give some guidance on which regions are the most promising in the (huge) space of solutions. This may be done by considering a simplified model or a relaxation that would provide some structural information about good solutions.

Acknowledgments The authors wish to thank the two anonymous reviewers for fruitful suggestions which help improve a previous version of this paper.

References

Bai, R., Blazewicz, J., Burke, E. K., Kendall, G., & McCollum, B. (2012). A simulated annealing hyper-heuristic methodology for flexible decision support. *4OR: A Quarterly Journal of Operations Research*, *10*(1), 43–66.

Bai, R., & Kendall, G. (2005). An investigation of automated planograms using a simulated annealing based hyper-heuristic. In T. Ibaraki, K. Nonobe & M. Yagiura (Eds.), *Metaheuristics: Progress as real problem solvers* (pp. 87–108). New York: Springer.

Bilgin, B., Özcan, E., & Korkmaz, E. E. (2006). An experimental study on hyper-heuristics and exam timetabling. In *Practice and theory of automated timetabling VI, 6th international conference, PATAT, Brno, Czech Republic, Revised selected papers* (pp. 394–412). doi:10.1007/978-3-540-77345-0_25.

Burke, E. K., Gendreau, M., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., et al. (2013). Hyper-heuristics: A survey of the state of the art. *Journal of the Operational Research Society*, *64*(12), 1695–1724.

Burke, E. K., Hyde, M., Kendall, G., Ochoa, G., Özcan, E., & Woodward, J. R. (2010). A classification of hyper-heuristic approaches. In M. Gendreau & J. -Y. Potvin (Eds.), *Handbook of metaheuristics* (pp. 449–468). New York: Springer.

Burke, E. K., Kendall, G., Misir, M., & Özcan, E. (2012). Monte carlo hyper-heuristics for examination timetabling. *Annals of Operations Research*, *196*(1), 73–90.

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

Cattrysse, D. G., & Van Wassenhove, L. N. (1992). A survey of algorithms for the generalized assignment problem. *European Journal of Operational Research*, *60*(3), 260–272.

Chen, Y., Das, A., Qin, W., Sivasubramaniam, A., Wang, Q., & Gautam, N. (2005). Managing server energy and operational costs in hosting centers. In *Proceedings of the ACM international conference on measurement and modeling of computer systems (SIGMETRICS)* (pp. 303–314). doi:10.1145/1064212.1064253.

Chung, F. R., Garey, M. R., & Johnson, D. S. (1982). On packing two-dimensional bins. *SIAM Journal on Algebraic Discrete Methods*, *3*(1), 66–76. doi:10.1137/0603007.

Cowling, P., Kendall, G., & Soubeiga, E. (2001). A hyperheuristic approach to scheduling a sales summit. In E. K. Burke & W. Erben (Eds.), *Practice and theory of automated timetabling III* (pp. 176–190). New York: Springer.

Crainic, T. G., & Gendreau, M. (2002). Cooperative parallel tabu search for capacitated network design. *Journal of Heuristics*, *8*(6), 601–627.

Crainic, T. G., Gendreau, M., Hansen, P., & Mladenović, N. (2004). Cooperative parallel variable neighborhood search for the p-median. *Journal of Heuristics*, *10*(3), 293–314.

Dowsland, K. A., Soubeiga, E., & Burke, E. K. (2007). A simulated annealing based hyperheuristic for determining shipper sizes for storage and transportation. *European Journal of Operational Research*, *179*(3), 759–774. doi:10.1016/j.ejor.2005.03.058.

Gavish, B., & Pirkul, H. (1991). Algorithms for the multi-resource generalized assignment problem. *Management Science*, *37*(6), 695–713. doi:10.1287/mnsc.37.6.695.

Han, B. T., Diehr, G., & Cook, J. S. (1994). Multiple-type, two-dimensional bin packing problems: Applications and algorithms. *Annals of Operations Research*, *50*(1), 239–261. doi:10.1007/BF02085642.

Hochbaum, D. S. (1996). *Approximation algorithms for NP-hard problems*. Boston, MA: PWS Publishing.

James, T., Rego, C., & Glover, F. (2009). A cooperative parallel tabu search algorithm for the quadratic assignment problem. *European Journal of Operational Research*, *195*(3), 810–826.

Kalender, M., Kheiri, A., Özcan, E., & Burke, E. K. (2013). A greedy gradient-simulated annealing hyper-heuristic. *Soft Computing*, *17*(12), 2279–2292.

Kellerer, H., Pferschy, U., & Pisinger, D. (2004). *Knapsack problems*. New York: Springer.

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science*, *220*(4598), 671–680.

Le Bouthillier, A., & Crainic, T. G. (2005). A cooperative parallel meta-heuristic for the vehicle routing problem with time windows. *Computers & Operations Research*, *32*(7), 1685–1708. doi:10.1016/j.cor.2003.11.023.

Lehre, P., & Özcan, E. (2013). A runtime analysis of simple hyper-heuristics: To mix or not to mix operators. In *Proceedings of the 12th ACM workshop on foundations of genetic algorithms* (pp. 97–104).

Lodi, A., Martello, S., & Monaci, M. (2002). Two-dimensional packing problems: A survey. *European Journal of Operational Research*, *141*(2), 241–252. doi:10.1016/S0377-2217(02)00123-6.

Martello, S., Pisinger, D., & Vigo, D. (2000). The three-dimensional bin packing problem. *Operations Research*, *48*(2), 256–267. doi:10.1287/opre.48.2.256.12386.

Maruyama, K., Chang, S., & Tang, D. (1977). A general packing algorithm for multidimensional resource requirements. *International Journal of Computer & Information Sciences*, *6*(2), 131–149. doi:10.1007/BF00999302.

Miyazawa, F. K., & Wakabayashi, Y. (2007). Two- and three-dimensional parametric packing. *Computers and Operations Research*, *34*, 2589–2603. doi:10.1016/j.cor.2005.10.001.

Nonobe, K., & Ibaraki, T. (2001). An improved tabu search method for the weighted constraint satisfaction problem. *INFOR: Information Systems and Operational Research*, *39*, 131–151.

Ouelhadj, D., & Petrovic, S. (2008). A cooperative distributed hyper-heuristic framework for scheduling. In *IEEE international conference on systems, man and cybernetics (SMC)* (pp. 2560–2565). IEEE.

Pentico, D. W. (2007). Assignment problems: A golden anniversary survey. *European Journal of Operational Research*, *176*(2), 774–793. doi:10.1016/j.ejor.2005.09.014.

Pisinger, D., & Ropke, S. (2007). A general heuristic for vehicle routing problems. *Computers & Operations Research*, *34*(8), 2403–2435.

Puchinger, J., & Raidl, G. R. (2007). Models and algorithms for three-stage two-dimensional bin packing. *European Journal of Operational Research*, *183*(3), 1304–1327. doi:10.1016/j.ejor.2005.11.064.

Rattadilok, P., Gaw, A., & Kwan, R. (2005). Distributed choice function hyper-heuristics for timetabling and scheduling. In E. Burke & M. Trick (Eds.), *Practice and theory of automated timetabling V, Lecture notes in computer science* (Vol. 3616, pp. 51–67). Berlin, Heidelberg: Springer. doi:10.1007/11593577_4.

Romeijn, H. E., & Morales, D. R. (2000). A class of greedy algorithms for the generalized assignment problem. *Discrete Applied Mathematics*, *103*(13), 209–235. doi:10.1016/S0166-218X(99)00224-3.

Ropke, S., & Pisinger, D. (2006). An adaptive large neighborhood search heuristic for the pickup and delivery problem with time windows. *Transportation Science*, *40*(4), 455–472. doi:10.1287/trsc.1050.0135.

Spieksma, F. C. R. (1994). A branch-and-bound algorithm for the two-dimensional vector packing problem. *Computers & Operations Research*, *21*(1), 19–25. doi:10.1016/0305-0548(94)90059-0.

Vazirani, V. V. (2001). *Approximation algorithms*. New York: Springer.

Wood, T., Shenoy, P. J., Venkataramani, A., & Yousif, M. S. (2007). Black-box and gray-box strategies for virtual machine migration. In *Proceedings of the 4th USENIX conference on Networked systems design and implementation (NSDI'07)* (Vol. 7, pp. 229–242).

Yagiura, M., Ibaraki, T., & Glover, F. (2004a). An ejection chain approach for the generalized assignment problem. *INFORMS Journal on Computing*, *16*(2), 133–151. doi:10.1287/ijoc.1030.0036.

Yagiura, M., Ibaraki, T., & Glover, F. (2006). A path relinking approach with ejection chains for the generalized assignment problem. *European Journal of Operational Research*, *169*(2), 548–569. doi:10.1016/j.ejor.2004.08.015.

Yagiura, M., Iwasaki, S., Ibaraki, T., & Glover, F. (2004). A very large-scale neighborhood search algorithm for the multi-resource generalized assignment problem. *Discrete Optimization*, *1*, 87–98. doi:10.1016/j.disopt.2004.03.005.

Yagiura, M., Yamaguchi, T., & Ibaraki, T. (1998). A variable depth search algorithm with branching search for the generalized assignment problem. *Optimization Methods and Software*, *10*, 419–441. doi:10.1080/10556789808805722.