Computers & Operations Research 68 (2016) 16–29

Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/caor

Multi-neighborhood local search optimization for machine reassignment problem

Zhuo Wang, Zhipeng Lü, Tao Ye*

School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan 430074, China

ARTICLE INFO

Available online 31 October 2015

Keywords:
Machine reassignment
Google ROADEF/EURO Challenge
Local search
Neighborhood structure
Partition search

ABSTRACT

As the topic of the Google ROADEF/EURO Challenge 2012, machine reassignment problem (denoted as MRP) is an important optimization problem in load balance of cloud computing. Given a set of machines and a set of processes running on machines, the MRP aims at finding a best process-machine reassignment to improve the usage of machines while satisfying various hard constraints. In this paper, we present a metaheuristic algorithm based on multi-neighborhood local search (denoted as MNLS) for solving the MRP. Our MNLS algorithm consists of three primary and one auxiliary neighborhood structures, an efficient neighborhood partition search mechanism with respect to the three primary neighborhoods and a dynamic perturbation operator. Computational results tested on 30 benchmark instances of the ROADEF/EURO Challenge 2012 and comparisons with the results in the challenge and the literature demonstrate the efficacy of the proposed MNLS algorithm in terms of both effectiveness and efficiency. Furthermore, several key components of our MNLS algorithm are analyzed to gain an insight into it.

1. Introduction

Cloud computing [1] has emerged as a new technology to support massive computing and storage task. A number of cloud computing services are currently being offered, such as Google Docs and Amazon Elastic Computing Cloud. A cloud computing platform is maintained by a data center which consists of a large pool of computing resources and storage devices shared by end users. As the services are expanding quickly, the issue of resource consumption of data centers is becoming increasingly important. Therefore, allocating computing resources and user requests in a cost-efficient way to improve the usage of servers plays a key role in cloud computing [2].

Aiming at finding an optimization solution to improve the usage of a set of machines, a challenging problem denoted by machine reassignment problem (MRP) has been proposed as the subject of the ROADEF/EURO Challenge 2012 by Google [3]. The MRP consists of reassigning processes among servers to improve the usage of machine resources while satisfying various constraints. The constraints which cannot be violated include capacity constraints, conflict constraints, spread constraints, dependency constraints and transient usage constraints. The objective function modeling the efficiency of the reassignment incorporates load cost, balance cost and three kinds of migration costs of processes.

For the MRP, several state-of-the-art algorithms have been proposed in the literature. Mehta et al. [4] introduced a constraint programming (CP) formulation of the problem, and proposed a CP-based large neighborhood search to address it. Malitsky et al. [5] further studied the impact of the parameters used in the proposed CP-based large neighborhood search. Gavranović et al. [6] proposed a variable neighborhood search which consists of four kinds of neighborhood structures. Brandt et al. [7] proposed a large neighborhood search algorithm that uses a constraint programming to find improving solutions. Masson et al. [8] put forward a multi-start iterated local search which explores two kinds of neighborhoods. Portal [9] proposed a randomized local search based on simulated annealing (SA) [10] for solving the MRP. Jaśkowski et al. [11] proposed a hybrid metaheuristic approach which consists of a fast greedy hill climber and a large neighborhood search. Portal et al. [12] put forward a simulated annealing algorithm using two neighborhoods to address the MRP. Lopes et al. [13] introduced a linear integer programming (IP) formulation for the MRP, and proposed an iterated local search algorithm for solving this problem.

The MRP belongs to the family of the assignment problem [14–16]. In particular, there is a similar theoretical assignment problem of MRP in the literature which is called generalized assignment problem (GAP) [17,18]. The GAP deals with a minimum cost

* Corresponding author.
E-mail addresses: 575880296@qq.com (Z. Wang), zhipeng.lv@hust.edu.cn (Z. Lü), yeetao@gmail.com (T. Ye).

http://dx.doi.org/10.1016/j.cor.2015.10.015
0305-0548/

assignment of a number of jobs to a set of agents such that every job is assigned to exactly one agent and the resource constraint for each agent is satisfied. It has many practical applications, such as vehicle routing [19–21] and facility location [22–24], and has been extensively studied in the literature. The approaches for GAP can be divided into two main categories: metaheuristic algorithms and exact algorithms.

The metaheuristic algorithms for solving the GAP include: a variable depth search algorithm by Amini and Racer [25]; a tabu search (TS) [26] and simulated annealing by Osman [27]; a genetic algorithm [28] by Chu and Beasley [29]; variable depth search algorithms by Yagiura et al. [30,31]; a guided genetic algorithm by Lau and Tsang [32]; an adaptive search heuristic based on greedy randomized adaptive search procedure (GRASP) [33] and Max-Min ant system (MMAS) [34] by Lourenço and Serra [35]; an ejection chain approach by Yagiura et al. [36]; a path relinking approach with ejection chains by Yagiura et al. [37]; differential evolution algorithms by Tasgetiren et al. [38]; a bees algorithm with an ejection chain neighborhood by Özbakir et al. [39] and so on.

Exact algorithms for GAP are mainly based on branch and bound framework, e.g., a branch and bound algorithm by solving a series of binary knapsack problems to determine the bounds by Ross and Soland [40], a new algorithm employing both column generation and branch and bound by Savelsbergh [41], a breadth-first branch and bound algorithm by Haddadi and Ouzia [42]. Woodcock and Wilson [43] proposed a hybrid algorithm combining both branch and bound and tabu search for solving the GAP.

In this paper, we propose a multi-neighborhood local search based metaheuristic algorithm, denoted as MNLS, for solving the machine reassignment problem. Our algorithm integrates several different neighborhood structures, two of which are usually employed in the literature for solving the GAP and the other two are tailored for the MRP. In addition, a neighborhood partition search mechanism and a dynamic perturbation operator are introduced to enhance the search efficiency of the MNLS.

The remaining part of the paper is organized as follows. Section 2 presents the problem description and a mathematical formulation of the MRP. In Section 3, we give the main framework and specific components of our MNLS algorithm in details. In Section 4, several key components of our algorithm are investigated. In Section 5, we present computational results of our MNLS algorithm and extensive comparisons with the results in both the challenge and the literature. The paper is concluded in Section 6.

2. Machine reassignment problem

2.1. Problem description

The machine reassignment problem considered in this paper consists of reassigning processes to machines in accordance with a given set of constraints. A machine has several kinds of resources such as CPU and RAM, and runs processes which consume these resources. Initially, each process is allocated to a machine. In order to improve the usage of machines, processes can be moved from one machine to another. Two types of constraints are defined: those which must be strictly satisfied under any circumstances (hard constraints) and those which are not necessarily satisfied but whose violations should be desirably minimized (soft constraints). Feasible moves are limited by hard constraints and have costs composed of violations of soft constraints. A reassignment that satisfies all hard constraints is called a feasible solution. The objective of this problem is to find a feasible solution which minimizes the total weighted violations of soft constraints.

Let $M=\{m_1, m_2, \ldots, m_k\}$ be the set of machines, and $P=\{p_1, p_2, \ldots, p_w\}$ the set of processes. A solution of this problem can be represented by a vector Map of length w, where $Map(p)$ corresponds to the machine assigned to process p. Additionally, let Map_0 be the given initial solution, so $Map_0(p)$ is the original machine assigned to process p correspondingly.

A number of constant and variable symbols are presented in Table 1. Note that *bool* is the truth indicator function which takes value of 1 if the given proposition is true and 0 otherwise. With these notations, we can describe the problem in a formal way.

The five hard constrains are:

- H_1. Capacity constraints. The usage of a machine for every resource cannot exceed its corresponding capacity.

$$\forall m \in M, r \in R, U(m,r) \le C(m,r)$$

- H_2. Conflict constraints. Processes of the same service must run on distinct machines.

$$\forall s \in S, (p_i, p_j) \in s^2, p_i \ne p_j \Rightarrow Map(p_i) \ne Map(p_j)$$

- H_3. Spread constraints. The number of locations where processes of the same service run cannot be less than a given minimum limit.

$$\forall s \in S, \sum_{l \in L} \min(1, |\{p \in s | Map(p) \in l\}|) \ge S\_\min(s)$$

- H_4. Dependency constraints. If a service s^a depends on a service s^b, i.e., $(s^a, s^b) \in SD$, then each process of s^a should run in the neighborhood of a s^b process.

$$\forall (s^a, s^b) \in SD, \forall p^a \in s^a, \exists p^b \in s^b \Rightarrow NE(Map(p^a)) = NE(Map(p^b))$$

- H_5. Transient usage constraints. When a process is moved from one machine to another, some resources are consumed twice. For example, disk space is occupied on both machines when a process is removed from one machine to another. Let $TR \subseteq R$ be the subset of resources which need transient usage, i.e., the processes using resources in TR require capacity on both the original machine and the current machine. The transient usage constraints are:

$$\forall m \in M, \mathrm{tr} \in TR, U(m,\mathrm{tr}) + TU(m,\mathrm{tr}) \le C(m,\mathrm{tr})$$

The five soft constraints are:

- S_1. Load cost. $SC(m,r)$ is the safety capacity of a resource $r \in R$ on a machine $m \in M$. The load cost is defined per resource and corresponds to the used capacity above the safety capacity:

$$f_1(r) = \sum_{m \in M} \max(0, U(m,r) - SC(m,r))$$

- S_2. Balance cost. As having available CPU resource without having available RAM resource is useless for future assignments, one objective of this problem is to balance available resources. The idea is to achieve a given target on the available ratio of two different resources [3]. $B \subseteq R^2 \times \mathrm{N}$ is the set of triples. For a given triple $b=(r_1^b, r_2^b, target^b) \in B$, the balance cost is:

$$f_2(b) = \sum_{m \in M} \max(0, target^b \cdot A(m, r_1^b) - A(m, r_2^b))$$

- S_3. Process move cost. Some processes are painful to move. To model this soft constraint a process move cost is defined:

Table 1
Constant and variable notations used for the MRP.

Symbols	Description
Constants	
M	The set of machines, $M=\{m_1,m_2,\ldots,m_k\}$, $\|M\|=k$
P	The set of processes, $P=\{p_1,p_2,\ldots,p_w\}$, $\|P\|=w$
R	The set of resources, $R=\{r_1,r_2,\ldots,r_d\}$, $\|R\|=d$
TR	The subset of resources which need transient usage, $TR=\{\mathrm{tr}_1,\mathrm{tr}_2,\ldots,\mathrm{tr}_h\}$, $\|TR\|=h$, $TR\subseteq R$
S	The set of services, $S=\{s_1,s_2,\ldots,s_f\}$, $\|S\|=f$. Processes are partitioned into services, all services are disjoint, i.e., $\cup_{s\in S}s=P$, $\forall s_i,s_j\in S, s_i\neq s_j, s_i\cap s_j=\varnothing$
L	The set of locations, $L=\{l_1,l_2,\ldots,l_g\}$, $\|L\|=g$. Machines are partitioned into locations, all locations are disjoint, i.e., $\cup_{l\in L}l=M$, $\forall l_i,l_j\in L, l_i\neq l_j, l_i\cap l_j=\varnothing$
N	The set of neighborhoods, $N=\{n_1,n_2,\ldots,n_v\}$, $\|N\|=v$. Machines are partitioned into neighborhoods, all neighborhoods are disjoint, i.e., $\cup_{n\in N}n=M$, $\forall n_i,n_j\in N, n_i\neq n_j, n_i\cap n_j=\varnothing$
SD	The set of service dependencies defined in S^2. $SD=\{sd_1,sd_2,\ldots,sd_z\}$, $\|SD\|=z$
B	The set of triples defined in $R^2\times$ N, N is the set of natural numbers. $B=\{b_1,b_2,\ldots,b_e\}$, $\|B\|=e$
$NE(m)$	The neighborhood of machine $m\in M$
$C(m,r)$	The capacity of resource $r\in R$ on machine $m\in M$
$R(p,r)$	The requirement of resource $r\in R$ for process $p\in P$
$S\_min(s)$	The minimum number of distinct locations where at least one process of service $s\in S$ should run
$SC(m,r)$	The safety capacity of resource $r\in R$ on machine $m\in M$
$PMC(p)$	The cost of moving process $p\in P$ from its original machine $Map_0(p)$
$MMC(m,m')$	The cost of moving any process $p\in P$ from machine $m\in M$ to machine $m'\in M$. $\forall m\in M, MMC(m,m)=0$
$wt_1(r)$	The weight of load cost for resource $r\in R$
$wt_2(b)$	The weight of balance cost for triple $b\in B$
wt_3, wt_4, wt_5	The weights of process move cost, service move cost and machine move cost
Variables	
$U(m,r)$	The usage of machine $m\in M$ for resource $r\in R$, i.e., $U(m,r)=\sum_{p\in P}bool(Map(p)=m)\cdot R(p,r)$
$A(m,r)$	The available capacity of resource $r\in R$ on machine $m\in M$, i.e., $A(m,r)=C(m,r)-U(m,r)$
$A(r)$	The total available capacity of resource $r\in R$ over all machines, i.e., $A(r)=\sum_{m\in M}A(m,r)$
$TU(m,r)$	The transient usage of machine $m\in M$ for resource $r\in R$, i.e., $\begin{cases}\sum_{p\in P}bool(Map_0(p)=m\wedge Map(p)\neq m)\cdot R(p,r), & r\in TR\\ 0, & r\in R-TR\end{cases}$
$CO(m)$	The capacity overload on machine $m\in M$, i.e., $CO(m)=\sum_{r\in R}\max(TU(m,r)-A(m,r),0)$

$$f_3=\sum_{p\in P}bool(Map_0(p)\neq Map(p))\cdot PMC(p)$$

- S_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:

$$f_4=\max_{s\in S}(|\{p\in s|Map(p)\neq Map_0(p)\}|)$$

- S_5. Machine move cost. The sum of all moves weighted by relevant *MMC*:

$$f_5=\sum_{p\in P}MMC(Map_0(p),Map(p))$$

With the above formulations, we can then calculate the total objective cost for a given candidate feasible solution *Map* according to the cost function $f(Map)$ defined in formula (1):

$$f(Map)=\sum_{r\in R}wt_1(r)\cdot f_1(r)+\sum_{b\in B}wt_2(b)\cdot f_2(b)+wt_3\cdot f_3+wt_4\cdot f_4+wt_5\cdot f_5 \quad (1)$$

where $wt_1(r)$ is the weight of load cost for resource r, $wt_2(b)$ is the weight of balance cost for triple b, wt_3, wt_4, wt_5 are respectively the weights of process move cost, service move cost and machine move cost. The goal is then to find a feasible solution Map^* such that $f(Map^*)\leq f(Map)$ for all *Map* in the feasible search space.

2.2. Lower bound

Here, we briefly present a simple lower bound (LB) described in [9] for the MRP which can be obtained in linear time relative to the problem size. It is an efficient method to evaluate the effectiveness of our proposed algorithm, while with the size of the problem getting larger, calculating the LB by linear programming becomes increasingly difficult.

The LB method ignores the process move cost, service move cost and machine move cost, and considers the load cost and balance cost independently. We give the formula of the LB directly without specific deduction. More details can be found in [9].

$$LB=\sum_{r\in R}wt_1(r)\cdot\max(0,\sum_{p\in P}R(p,r)-\sum_{m\in M}SC(m,r))+\sum_{b\in B}wt_2(b)\cdot\max(0,target^b\cdot A(r_1^b)-A(r_2^b)) \quad (2)$$

where $A(r)$ is the total available capacity of resource r over all machines.

3. Solution method

3.1. Main framework

Starting from the given initial feasible solution Map_0, our MNLS algorithm (Algorithm 1) launches the local search procedure to optimize the objective function as far as possible (line 6, see Section 3.4). In our local search procedure, three kinds of neighborhood structures (with an auxiliary one) are sequentially explored, including one process shift (N_1), two processes swap (N_2) and three processes swap (N_3), where a neighborhood partition technique is employed to partition the original neighborhood into smaller ones in order to enhance the search efficiency. When a round of local search stops, the best solution found so far will be perturbed as the initial solution of the next round of local search (line 10, see Section 3.5). After each round of local search, an important parameter *imth* used in the local search procedure is updated (lines 4 and 11, see Section 3.6). When the running time reaches a predetermined limit, the algorithm stops and returns the

best solution found so far. In the following subsections, the main components of our MNLS algorithm are described in details.

Algorithm 1. Main framework of the proposed MNLS algorithm.

1. **Input:** Problem instance I, Map_0
2. **Output:** The best solution Map^* found so far
3. $Map^* \leftarrow Map_0$, $Map \leftarrow Map_0$
   // Map_0 (Map) denotes the initial (current) solution
4. Initialize parameter $imth$
5. **While** (time limit is not reached) **do**
6. $Map' \leftarrow$ **Local_Search**(Map, $N_1 \rightarrow N_2 \rightarrow N_3$) /* Section 3.4*/
7. **If** ($f(Map') < f(Map^*)$) **then**
8. $Map^* \leftarrow Map'$
9. **End if**
10. $Map \leftarrow$ **Perturbation**(Map^*) /* Section 3.5*/
11. **Parameter_Updating**($imth$) /* Section 3.6*/
12. **End while**
13. **Return** Map^*

3.2. Moves and neighborhood structures

In a local search algorithm, applying a move mv to the current solution Map, we can obtain a new neighboring solution denoted by $Map \oplus mv$. Let $MV(Map)$ denote the set of moves which can be applied to Map, then the neighborhood of Map is defined by

$$N(Map) = \{Map \oplus mv | \, mv \in MV(Map)\} \tag{3}$$

In this paper, we use three main kinds of neighborhoods respectively defined by moving one process to another machine (one-shift), swapping two processes assigned to two different machines (two-swap) and swapping two processes on one machine with one process on another machine (three-swap). The former two are common in the literature for solving the GAP.

It is noteworthy that moves breaking the feasibility of the current solution are also considered in our three neighborhoods. For this purpose, we introduce an auxiliary neighborhood move (called repair move) to repair an infeasible neighboring solution into a feasible one, which is realized by moving some processes on one *bottleneck* machine to other machines. In the following, we first describe the three main neighborhoods and then the auxiliary one in details.

Specifically, a one-shift move applied to solution Map, denoted by $mv_1(p, m)$, means assigning machine m to process p. Formally:

$$MV_1(Map) = \{mv_1(p, m) | \, \forall p \in P, m \in M, Map(p) \neq m\} \tag{4}$$

Applying a two-swap move to solution Map, denoted by $mv_2(p_i, p_j)$, means assigning machine $Map(p_j)$ to process p_i, and inversely machine $Map(p_i)$ to process p_j. Formally:

$$MV_2(Map) = \{mv_2(p_i, p_j) | \, \forall p_i, p_j \in P, Map(p_i) \neq Map(p_j)\} \tag{5}$$

Applying a three-swap move to solution Map, denoted by $mv_3(p_i, p_j, p_k)$, means assigning machine $Map(p_i)$ to process p_k, and inversely machine $Map(p_k)$ to both process p_i and process p_j. Formally:

$$\begin{aligned} MV_3(Map) = \{&mv_3(p_i, p_j, p_k) | \, \forall p_i, p_j, p_k \in P, Map(p_i) \\ &= Map(p_j) \wedge Map(p_i) \neq Map(p_k)\} \end{aligned} \tag{6}$$

For convenience, we denote the neighborhoods represented by the move sets MV_1, MV_2 and MV_3 as N_1, N_2 and N_3, respectively.

In order to enhance the search efficiency of both N_2 and N_3, we employ two additional strategies to tailor the neighborhoods. The first one is to only randomly select at most 10 processes on each involved machine. The second is to exclude some non-promising pairs of machines. Specifically, for two machines m_i and m_j, we can obtain the lower bound of the load cost and balance cost of these two machines by formula (2) as described in Section 2.2. If the lower bound is equal to the sum of the load cost and balance cost of the two machines, the probability of finding a good swap move on these two machines is quite small. In this case, we do not consider to swap any pair of processes on these two machines.

As mentioned above, besides all feasible moves, our neighborhoods N_1, N_2 and N_3 also consider infeasible moves. Specifically, if a neighboring solution of the three neighborhoods violates capacity constraints (H_1) or transient usage constraints (H_5), we identify the violated (also called bottleneck) machine and try to move some processes away from this machine to recover the feasibility of the solution. This auxiliary move is called repair move.

Specifically, for a solution Map, let us define the capacity overload on machine $m \in M$ as:

$$CO(m) = \sum_{r \in R} \max(TU(m, r) - A(m, r), 0) \tag{7}$$

If the value of $CO(m)$ is positive, machine m is identified as the bottleneck machine. For this machine, we devise two different strategies to move a process on it away each time to decrease the capacity overload. The first one (Algorithm 2) is to find the best move in terms of the total cost function to move one process on this machine to another machine. The second one (Algorithm 3) is to select the process whose removal makes the capacity overload reduce the most, and then move it to the best machine in terms of the total cost function. Both repair procedures are repeated until the capacity overload value becomes 0 or no feasible move exists. Notice that during the process new violations are absolutely prohibited. If the repair does not succeed, we just discard this neighboring solution. It can be observed that the objective cost is the only factor considered in the first strategy, and the decrement of capacity overload is preferred over the objective cost in the second strategy. Preliminary experiments show that combining these two strategies is better than just using a single one. Therefore, we choose one of these two strategies randomly with equal probability to repair the bottleneck machine each time.

Algorithm 2. The first repair strategy.

1. **Input:** Map, m: the bottleneck machine
2. **Output:** if the repair succeeds
3. **While**($CO(m) > 0$)
4. $mv \leftarrow$ best feasible one-shift move that moves a process on m to another machine
5. **If**(such mv exists) **then**
6. $Map \leftarrow Map \oplus mv$
7. **Else**
8. **Return false**
9. **End if**
10. **End while**
11. **Return true**

Algorithm 3. The second repair strategy.

1. **Input:** Map, m: the bottleneck machine
2. **Output:** if the repair succeeds
3. **While**($CO(m) > 0$)
4. $p \leftarrow$ the process on m whose removal makes $CO(m)$ reduce the most
5. $mv \leftarrow$ best feasible one-shift that moves p to another machine
6. **If**(such mv exists) **then**
7. $Map \leftarrow Map \oplus mv$
8. **Else**
9. **Return false**

10. **End if**
11. **End while**
12. **Return true**

In this way, this auxiliary move greatly expands the search space of our original neighborhoods N_1, N_2 and N_3. That is to say, we actually use three augmented neighborhoods of one-shift, two-swap and three-swap, which is one of the most essential ingredients of our MNLS algorithm.

3.3. Neighborhood partition

In our local search procedure, we employ a neighborhood partition technique to partition neighborhoods N_1, N_2 and N_3 into smaller ones in order to reach a tradeoff between search effectiveness and efficiency. In such a way, our local search only explores part of the original neighborhood at each iteration. In our implementation, all the three neighborhoods N_1, N_2 and N_3 are equally and randomly partitioned. The underlying rationale to use this neighborhood partition technique lies in the fact that the original neighborhood sizes of N_1, N_2 and N_3 are quite huge for large-scale instances and we expect our local search to just examine a small part of the neighborhood at each iteration and thus to obtain fast improvement during the first iterations.

Specifically, for neighborhood N_1, we randomly partition all the processes P into a given number q_1 of parts with equal size, where the ith part is denoted as P_i. Thus, the ith partition of the neighborhood N_1, denoted as $N_1^{(i)}$, consists of all one-shift moves, whose moved processes belong to P_i.

For neighborhood N_2 (N_3), since each swap move involves two machines we randomly partition all the machines M into a given number q_2 (q_3) of parts with equal size, where the ith part is denoted as M_i. Thus, the ith partition of the neighborhood N_2 (N_3), denoted as $N_2^{(i)}$ ($N_3^{(i)}$), consists of all two-swap (three-swap) moves, whose involved two machines belong to M_i.

It is noteworthy that the union of the q_1 partitions is equal to the complete neighborhood N_1, i.e., $\cup_{i=1}^{q_1} N_1^{(i)} = N_1$, while the union of all the partitions is just a subset of the complete neighborhood N_2 or N_3, i.e., $\cup_{i=1}^{q_2} N_2^{(i)} \subseteq N_2$, $\cup_{i=1}^{q_3} N_3^{(i)} \subseteq N_3$. The reason is that the processes of the machines belonging to different partitions would not be swapped.

3.4. Local search procedure

Based on the above-mentioned neighborhood definition and partition technique, we now describe our local search procedure in details. The following subsections present the multi-neighborhood search strategy, the local search procedure, the move selection heuristic and the termination criterion, respectively.

3.4.1. Multi-neighborhood search strategy

Our multi-neighborhood local search procedure is described in Algorithm 4. In the procedure, we sequentially explore the three neighborhoods N_1, N_2 and N_3 (lines 3–5). That is to say, neighborhood N_1 is first examined. Once it reaches a local optimum trap, neighborhood N_2 is employed to further optimize the objective function. When it detects stagnation too, the last neighborhood N_3 is finally utilized.

Notice that the local search procedure for each single neighborhood is only performed once in each round. Moreover, the initial solution of the current neighborhood search N_2 (N_3) is just the best solution found by the previous neighborhood N_1 (N_2). This multi-neighborhood search process is denoted as $N_1 \rightarrow N_2 \rightarrow N_3$ in Algorithm 1.

Algorithm 4. Multi-neighborhood local search procedure.

1. **Input:** *Map*: initial solution, N_1, N_2, N_3: neighborhoods
2. **Output:** *Map*′: best solution found in a round of local search
3. *Map* ← **Local_Search**(*Map*, N_1) /* Section 3.4.2*/
4. *Map* ← **Local_Search**(*Map*, N_2) /* Section 3.4.2*/
5. *Map*′ ← **Local_Search**(*Map*, N_3) /* Section 3.4.2*/
6. **Return** *Map*′

3.4.2. Local search for a single neighborhood

Our local search procedure for a single neighborhood $N_i (i = 1, 2, 3)$ is described in Algorithm 5. Based on the partitioned neighborhood, our local search sequentially explores each part of neighborhood N_i (lines 6–7) and selects a move in the neighborhood $N_i^{(j)}$ at each iteration (Algorithm 6). This process is repeated until the termination criterion is satisfied.

Algorithm 5. Local search procedure for neighborhood N_i.

1. **Input:** *Map*: initial solution, N_i: neighborhood
2. **Output:** *Map*′: best solution found in local search
3. *Map*′ ← *Map*, *iter* ← 0 // *iter* denotes the iterations in local search
4. **While** (termination criterion is not met) **do** /* Section 3.4.4*/
5. **Neighborhood_Partition**(N_i) /* Section 3.3*/
6. **For** j:=1 **to** q_i **do**
7. $mv \leftarrow$ **move_selection_heuristic** (*Map*, $N_i^{(j)}$) /* Section 3.4.3*/
8. *Map* ← *Map* ⊕ mv
9. *iter* ← *iter* + 1
10. **If** (f(*Map*) < f(*Map*′)) **then**
11. *Map*′ ← *Map*
12. **End if**
13. **End for**
14. **End while**
15. **Return** *Map*′

3.4.3. Neighborhood move selection heuristic

In order to enhance the search capability of our algorithm, we introduce a move selection heuristic which includes a set of key components, including a tabu table to avoid selecting the recently visited moves, a condition to accept infeasible moves and a random move strategy. All these components are jointly used by our move selection heuristic, as described in Algorithm 6.

Tabu search (TS) [26] typically incorporates a tabu list as "recency based" memory structure to assure that solutions visited within a certain span of iterations, called the tabu tenure, will not be revisited. Our MNLS algorithm uses such a tabu list as one of its diversification strategies. In our implementation, each time a process p is moved away from a machine, a value is assigned to an associated record *TabuTenure*(p) (identifying the "tabu tenure") to prevent p from being moved again for the next *TabuTenure*(p) iterations. A neighborhood move is called tabu if all the involved processes are in tabu status. Because of the flexibility of the neighborhood move selection heuristic, the performance of our algorithm is not very sensitive to the value of tabu tenure. Therefore, we set the parameter $TabuTenure(p) = |P|/100$.

The move selection strategy works as follows for neighborhood $N_i^{(j)}$. First, we employ the standard tabu search strategy that restricts consideration to non-tabu feasible moves in $N_i^{(j)}$ (line 6). However, an aspiration criterion is applied that permits a feasible move to be selected in spite of being tabu if it leads to a solution better than the best solution *Map*′ (line 4). If the current solution can be improved

after applying the move, i.e., $f(Map \oplus mv) - f(Map) < 0$, we accept this move (lines 8–9). Otherwise, we consider the best infeasible move in $N_i^{(j)}$ (line 11). If this best infeasible move can be repaired by our auxiliary move, we will accept this repaired move (lines 12–13). Otherwise, we just randomly select a feasible move regardless of the objective cost in the whole neighborhood N_i.

Algorithm 6. Neighborhood move selection heuristic for $N_i^{(j)}$.

1. **Input:** Map, $N_i^{(j)}$
2. **Output:** mv
3. **If** (aspiration criterion is satisfied) **then**
4. $mv \leftarrow$ best feasible move in $N_i^{(j)}$
5. **Else**
6. $mv \leftarrow$ best feasible non-tabu move in $N_i^{(j)}$
7. **End if**
8. **If** $(f(Map \oplus mv) - f(Map) < 0)$ **then**
9. **Return** mv
10. **Else**
11. $mv \leftarrow$ best infeasible move in $N_i^{(j)}$
12. **If** (mv can be repaired) **then**
13. **Return** repaired mv
14. **Else**
15. $mv \leftarrow$ a random feasible move in N_i
16. **Return** mv
17. **End if**
18. **End if**

3.4.4. Termination criterion

Our local search procedure adopts the same termination criterion for all the three neighborhoods. Specifically, our local search stops if the improvement ratio of the best solution Map', i.e., $\frac{f(Map'') - f(Map')}{f(Map'')} \times 100\%$, does not exceed a given value *imth* during the last 100 local search iterations, where Map'' represents the best solution 100 iterations ago. The parameter *imth* is initialized with a value, and is gradually decreased after a round of local search. Details can be found in Section 3.6.

3.5. Perturbation operator

After a round of local search, we employ a perturbation operator in order for the search to escape from local optimum trap. Usually, perturbation is to apply some potentially-deteriorating moves which are less likely to be selected in local search. In our algorithm, a combination of one-shift, two-swap and three-swap moves is used.

At each iteration of our perturbation operator, we perform a random feasible move of the three neighborhoods. Specifically, at each iteration a one-shift move is applied with probability 0.5, a two-swap move is applied with probability 0.2, and a three-swap move is applied with probability 0.3. This process is repeated for a given number of times, called perturbation strength. We empirically set our perturbation strength to be $|P|/str$, where *str* is a random integer in interval [15, 20].

3.6. Parameter updating

Previous research has demonstrated that it is highly desirable to have a mechanism to set and adjust the parameters such that a near optimal performance can be achieved. As described above, an important parameter affecting the performance of our local search procedure is *imth*.

The *imth* value represents the stop criterion of our local search. One observes that the smaller the value of *imth* is, the more moves are applied in the local search, and the higher possibility that the search can intensively explore a neighborhood. However, it is obvious that it will need more time in a local search procedure. Therefore, we need to set an appropriate value to this parameter and adjust it according to the search history.

At the beginning of the search, due to the ease of improvement, parameter *imth* is first set at a relatively high level, and it is gradually decreased since improving the objective function value becomes more and more difficult with the search progressing. Specifically, after each local search phase, *imth* is decreased by $imth = 0.9 \times imth$.

In sum, the dynamic adjustment of the parameter allows the search to get a better tradeoff between efficiency and effectiveness.

4. Analysis and discussion

4.1. Benchmark instances

The benchmark dataset of the MRP consists of 30 instances of 3 sets used in the ROADEF/EURO Challenge 2012, where each set of A, B and X consists of 10 instances. The size of instances in Set A ranges from 4 machines and 100 processes to 100 machines and 1000 processes, and the size of Set B and X ranges from 100 machines and 5000 processes to 5000 machines and 50,000 processes. Table 2 shows the main characteristics of these instances. In the columns, the instance name, numbers of resources, resources needing transient usage, machines, processes, services, locations, neighborhoods, balance triples and service dependencies are reported.

4.2. Experimental protocol

The MNLS algorithm is programmed in C++ and compiled using GNU GCC. All tests are conducted on a server running

Table 2
The list of benchmark instances with their characteristics.

Inst.	$\|R\|$	$\|TR\|$	$\|M\|$	$\|P\|$	$\|S\|$	$\|L\|$	$\|N\|$	$\|B\|$	$\|SD\|$
a1_1	2	0	4	100	79	4	1	1	0
a1_2	4	1	100	1000	980	4	2	0	40
a1_3	3	1	100	1000	216	25	5	0	342
a1_4	3	1	50	1000	142	50	50	1	297
a1_5	4	1	12	1000	981	4	2	1	32
a2_1	3	0	100	1000	1000	1	1	0	0
a2_2	12	4	100	1000	170	25	5	0	0
a2_3	12	4	100	1000	129	25	5	0	577
a2_4	12	0	50	1000	180	25	5	1	397
a2_5	12	0	50	1000	153	25	5	0	506
b_1	12	4	100	5000	2512	10	5	0	4412
b_2	12	0	100	5000	2462	10	5	1	3617
b_3	6	2	100	20,000	15,025	10	5	0	16,560
b_4	6	0	500	20,000	1732	50	5	1	40,485
b_5	6	2	100	40,000	35,082	10	5	0	14,515
b_6	6	0	200	40,000	14,680	50	5	1	42,081
b_7	6	0	4000	40,000	15,050	50	5	1	43,873
b_8	3	1	100	50,000	45,030	10	5	0	15,145
b_9	3	0	1000	50,000	4609	100	5	1	43,437
b_10	3	0	5000	50,000	4896	100	5	1	47,260
x_1	12	4	100	5000	2529	10	5	0	4164
x_2	12	0	100	5000	2484	10	5	1	3742
x_3	6	2	100	20,000	14,928	10	5	0	15,201
x_4	6	0	500	20,000	1190	50	5	1	38,121
x_5	6	2	100	40,000	34,872	10	5	0	20,560
x_6	6	0	200	40,000	14,504	50	5	1	39,890
x_7	6	0	4000	40,000	15,273	50	5	1	43,726
x_8	3	1	100	50,000	44,950	10	5	0	12,150
x_9	3	0	1000	50,000	4871	100	5	1	45,457
x_10	3	0	5000	50,000	4615	100	5	1	47,768

Windows Server 2012 with Intel Xeon E5-2609 2.5 GHz and 32 GB of RAM. Our algorithm uses a single core only for each run and its default runtime is set to 300 s which is the time limit of the ROADEF/EURO Challenge 2012. The initial value of *imth* is set to 0.0007 which is determined by testing 15 different values in the range [0.00001, 0.01].

4.3. Influence of neighborhood partition

Considering the sizes of some large instances, we employ a neighborhood partition technique to partition the original neighborhood into smaller ones. As described above, we randomly partition all the processes P into q_1 parts of equal size for neighborhood N_1, and all the machines M into q_2 (q_3) parts for neighborhood N_2 (N_3) respectively.

In order to evaluate the influence of this mechanism and find proper values of q_1, q_2 and q_3, we conduct comparative experiments to test the partition of each neighborhood independently. For each neighborhood, we adopt four different settings of the corresponding parameter, and test them on instance b_10 for it is the largest and one of the most difficult instances. Notice that similar phenomenon can be observed on other instances. Our MNLS algorithm is run for 10 times independently in each case. Tables 3–5 show the settings of the parameter and the computational results for N_1, N_2 and N_3, respectively. Column *iter* represents the total number of local search iterations, column f_{best} gives the obtained best objective value, and column f_{avr} gives the average objective value (averaged over 10 independent runs). It can be seen from the tables that all the three neighborhoods are not partitioned in case 1, and are partitioned into the most parts in case 4. The setting in case 1 obtains the worst solutions, which means the performance of the algorithm is quite poor for large instance if the original neighborhoods are not partitioned. It is obvious that the more partitions there are, the more local search iterations are performed, which means that partition is beneficial to the search efficiency. But the solution will get worse when the number of partitions is too large since the quality of the selected move becomes poor due to the very limited range of the candidates at each iteration.

To demonstrate the influence of proper neighborhood partition further, Fig. 1 shows how the best objective value and the local search iterations evolve with the CPU time in the above 4 cases for N_1. From Fig. 1, one finds that the speed of improvement in case 3 is the fastest although its speed of local search is not. The speed of improvement will get slower if the number of partitions is too large or small.

According to this comparison, we find that the speed of local search is acceptable if the size of the neighborhood explored at each iteration does not exceed the magnitude of 10^6. When the size of the partitioned neighborhood is smaller than the magnitude of 10^5, the solution quality will get worse although the speed of local search becomes faster. Therefore, we empirically set $q_1 = \max(1, |P| \times |M|/10^5)$, $q_2 = \max(1, |M|/100)$ and $q_3 = \max(1, |M|/50)$.

Table 3
Settings of parameter of partition for N_1.

case	$\lvert P\rvert/q_1$	q_1	*iter*	f_{best}	f_{avr}
case 1	50,000	1	19	41,001,630,159	41,001,636,168
case 2	200	250	4140	18,876,800,168	18,947,030,871
case 3	20	2500	22,934	18,769,858,060	18,979,609,048
case 4	2	25,000	46,034	23,018,235,325	24,369,406,656

Table 4
Settings of parameter of partition for N_2.

case	$\lvert M\rvert/q_2$	q_2	*iter*	f_{best}	f_{avr}
case 1	5000	1	2	41,969,924,876	41,987,036,267
case 2	1000	5	54	39,351,057,930	39,521,162,824
case 3	100	50	7263	18,124,024,469	18,131,515,717
case 4	10	500	768,127	18,276,842,548	18,363,388,623

Table 5
Settings of parameter of partition for N_3.

case	$\lvert M\rvert/q_3$	q_3	*iter*	f_{best}	f_{avr}
case 1	5000	1	1	42,105,172,633	42,117,107,826
case 2	500	10	33	39,657,209,444	39,872,188,397
case 3	50	100	3700	18,093,623,026	18,195,937,267
case 4	5	1000	554,699	18,782,928,692	18,866,598,567

4.4. Significance of neighborhoods

Our algorithm explores three kinds of neighborhood structures sequentially during the local search procedure. In order to ensure that these neighborhood searches play a meaningful role, we conduct experiments to present and analyze performances of our algorithm in different cases. Specifically, we test our algorithm under the condition without or with only one kind of neighborhoods and compare the results with those obtained by the complete algorithm. The experimental protocol is the same as above.

Fig. 2 shows how the best objective value (averaged over 10 independent runs) evolves with the CPU time under different conditions on instance a1_4. The curve labeled with f_{best} represents the result obtained by the complete algorithm. The curves labeled with MV_1, MV_2 and MV_3 represent the case with only one-shift local search, two-swap local search and three-swap local search, respectively, and those labeled with $\overline{MV_1}$, $\overline{MV_2}$ and $\overline{MV_3}$ represent the result obtained under the condition without one-shift local search, two-swap local search and three-swap local search, respectively.

It can be seen from Fig. 2 that the result of the complete algorithm outperforms the results of other 6 combinations. The second best combination is the algorithm without two-swap local search, and the worst one is the algorithm with only one-shift local search.

To observe the overall comparison, we summarize in Table 6 the average differential ratios of the results of the above 6 different combinations of neighborhood structures with respect to the results of the complete one.

From Table 6, we can find that the algorithm exploring three kinds of neighborhoods obtains the best solutions in general. The differential ratio of the incomplete combination is rather large for some instances, such as a2_1, x_3, x_5 and x_8. This experiment provides an empirical justification of exploring all the three neighborhoods simultaneously.

4.5. Importance of infeasible moves

As mentioned above, it is allowed to employ infeasible moves in the local search. Once an infeasible move is applied, we try to perform a repair move to recover the feasibility of the solution. In order to evaluate the importance of infeasible moves, we compare two versions of our MNLS algorithm with and without applying infeasible moves.

The average differential ratios of the results of the algorithm without infeasible moves with respect to the results with infeasible

moves are presented in Table 7, where columns 2, 4 and 6 give the ratios respectively.

From Table 7, one finds that the complete algorithm obtains quite competitive results for most instances. The algorithm with infeasible moves gets better results than those without infeasible moves for 14 out of the 30 instances, while obtaining slightly worse results only for 4 instances. The maximum improvement ratio of the algorithm without infeasible moves over the complete MNLS algorithm is only 0.24% (for instance b_3). Moreover, there are 10 instances for which the differential ratios are larger than 1% (a1_4, a2_1, a2_2, a2_3, a2_5, b_1, x_1, x_3, x_5 and x_8), implying that the strategy of applying infeasible moves is essential for solving these instances.

To deeply explore the influence of the infeasible moves, we study some more detailed information about the comparison. Fig. 3 presents how the current objective value (left) and the best objective value (right) evolve with the CPU time. The curves labeled with f_{cur} and f_{best} represent the results obtained with applying infeasible moves, and the curves labeled with f'_{cur} and f'_{best} represent the results obtained without applying infeasible moves.

It can be observed that the complete algorithm obtains better results in terms of both the current and the best objective values. Specifically, the performance of the algorithm without infeasible moves is competitive in the initial stage, but it can hardly continue improving the solution when the search progresses. It is also noteworthy that there are more "jumps" of the current objective value in the search process of the complete algorithm, which indicates that infeasible moves bring more diversification into the search. In our algorithm, the search is trapped in local optima when there are not feasible moves which can improve the current solution. An infeasible move and the following repair move reassign a number of processes at each iteration. The more processes are moved, the more drastically the current reassignment is changed. This experiment demonstrates that the resulting diversification can help the search to jump out of the local optimum trap.

In addition, we analyze the number of processes that are finally moved from their original machines under two circumstances with and without infeasible moves. Specifically, we sort the processes by the total requirements for all the resources in a descending order and partition them into five equal parts (top 20%, 20–40%, 40–60%, 60–80% and bottom 20%), then count the number of processes that are finally reassigned for each part and calculate corresponding proportion respect to the total processes of each part. Table 8 shows this comparison. Columns 3 to 7 respectively present the percentages of reassigned processes for these five parts. The last column reports the percentage of all the reassigned processes. For each instance, the upper row is obtained with applying infeasible moves, and the lower one is obtained without infeasible moves, respectively. The last two rows report the average results.

From the last column in Table 8, one finds that there are more processes reassigned in the case of applying infeasible moves. In addition, the numbers of reassigned processes with infeasible moves on the first part, i.e., the first 20% largest processes, are more than those without infeasible moves for all the four

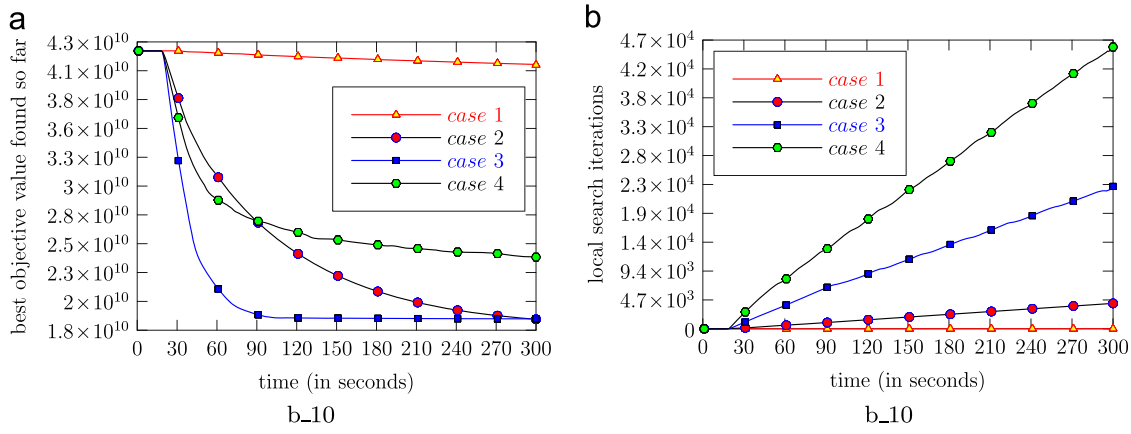

Fig. 1. Influence of neighborhood partition. The curves show how the best objective value and the local search iterations evolve with time. (a) b_10, (b) b_10.

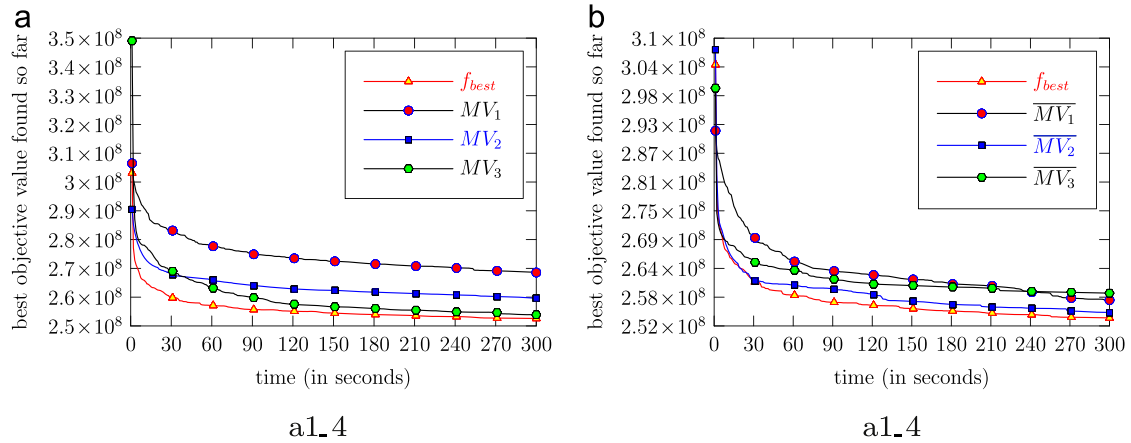

Fig. 2. Significance of neighborhoods. The curves show how the best objective value evolves with time. (a) a1_4, (b) a1_4.

Table 6
Comparison with different combinations of neighborhood structures.

Inst.	ΔMV_1(%)	ΔMV_2(%)	ΔMV_3(%)	$\Delta\overline{MV_1}$(%)	$\Delta\overline{MV_2}$(%)	$\Delta\overline{MV_3}$(%)
a1_1	0.00	0.00	0.00	0.00	0.00	0.00
a1_2	0.22	1.08	0.60	1.12	0.50	0.04
a1_3	0.00	0.00	0.00	0.00	0.00	0.00
a1_4	6.36	2.80	0.52	1.42	0.42	1.97
a1_5	0.00	0.00	0.00	0.00	0.00	0.00
a2_1	2,971,180.99	1,859,655.79	149,848.35	192,937.60	2.07	415,534.71
a2_2	6.11	14.61	4.66	6.74	−1.26	0.57
a2_3	2.90	10.39	4.76	6.41	−1.44	2.52
a2_4	0.47	0.24	0.02	0.01	0.01	0.04
a2_5	10.38	3.49	1.52	0.48	0.71	2.31
b_1	3.19	2.08	2.41	1.93	0.24	0.01
b_2	7.05	0.23	−0.31	−0.07	0.06	−0.05
b_3	29.59	47.03	2.28	12.82	1.54	−0.27
b_4	0.01	0.00	0.00	0.00	0.00	0.00
b_5	4.48	15.02	9.57	15.00	0.00	0.01
b_6	0.00	0.00	0.00	0.00	0.00	0.00
b_7	3.68	3.92	8.09	3.96	0.01	0.02
b_8	0.02	1.09	0.39	0.61	−0.01	0.00
b_9	0.03	0.00	0.00	0.00	0.01	0.00
b_10	5.37	0.39	0.21	0.27	0.19	0.00
x_1	1.73	2.45	2.85	1.92	−0.06	−0.21
x_2	5.56	−0.42	−0.60	−0.24	−0.35	0.38
x_3	5200.26	13,725.16	35.31	1749.66	25.85	−2.25
x_4	0.03	0.00	0.00	0.00	0.00	0.00
x_5	3719.08	28,083.90	13,232.88	28,177.09	4.01	−0.40
x_6	0.00	0.00	0.00	0.00	0.00	0.00
x_7	5.65	5.16	10.56	5.33	−0.12	−0.01
x_8	1.95	31,539.83	4302.37	11,583.48	17.02	17.93
x_9	0.03	0.00	0.00	0.00	0.00	0.00
x_10	5.50	0.42	0.23	0.29	0.17	0.01

Table 7
Comparison with the algorithm without infeasible moves.

Inst.	$\Delta f'$(%)	Inst.	$\Delta f'$(%)	Inst.	$\Delta f'$(%)
a1_1	0.00	b_1	2.60	x_1	1.09
a1_2	0.47	b_2	0.33	x_2	0.23
a1_3	0.00	b_3	−0.24	x_3	9.35
a1_4	1.12	b_4	0.00	x_4	0.00
a1_5	0.00	b_5	0.00	x_5	7.28
a2_1	212,751.24	b_6	0.00	x_6	0.00
a2_2	8.73	b_7	0.00	x_7	−0.09
a2_3	3.93	b_8	−0.01	x_8	10.95
a2_4	0.43	b_9	0.00	x_9	0.00
a2_5	14.90	b_10	−0.01	x_10	0.00

instances, which implies that applying infeasible moves makes large-size processes easier to move. According to our observation, we need to move a certain number of large processes to high-capacity machines for a better reassignment. As moving large processes is more likely to violate capacity constraints, it is hard to perform this kind of moves without permission of infeasible intermediate solutions. This experiment further demonstrates the significance of the infeasible moves.

5. Computational results

5.1. Comparison with the best-known results

In order to assess the effectiveness of our MNLS algorithm, we perform 100 independent runs for each instance under 300 s timeout condition, and compare our results with the best-known results in the literature. Table 9 shows the computational results. Column 2 gives the initial solution cost. Column 3 gives the best-known cost in the literature. The remaining columns give the results of our algorithm according to five criteria: (1) the best objective value (f_{best}), (2) the average objective value (f_{avr}), (3) the differential ratio between our best solution and the best-known solution (r(%)), i.e., $\frac{f_{best}-\mathrm{BKS}}{\mathrm{BKS}}\times 100\%$, (4) the CPU time, t_{best} (in seconds), for reaching the best objective value f_{best}, and (5) the coefficient of variance (c_v(%)).

Table 9 discloses that our MNLS algorithm obtains good results in general. Specifically, our algorithm reaches the best results for 3 instances (a1_1, a1_3 and a1_5). For the remaining instances, the gaps are less than 1% for 18 instances, and are very close to 0 for 7 cases (a1_2, b_4, b_6, b_9, x_4, x_6 and x_9). One observes that the coefficients of variance are not greater than 1% for 21 instances, showing that our MNLS algorithm is quite robust.

To further assess the search potential of our algorithm, we prolong the runtime to 1 h and perform 20 independent runs for each instance. The computational results are presented in Table 10. From Table 10, it can be observed that our algorithm finds new upper bound for instance x_1 and gets competitive results for 17 instances whose gaps relative to the best-known solutions are less than 0.5%. Our best results can be further improved in 19 cases under this relaxed timeout condition, showing the search potential of our algorithm. In addition, the coefficients of variance are less than 1% for 23 instances and less than 0.5% for 20 cases, indicating that our algorithm becomes more robust when the runtime is prolonged.

5.2. Comparison with the results in the challenge

The specific results of each competitor on the Set B and X have been published by the challenge organizer. To compare our MNLS algorithm with the best methods in the challenge, we present the comparison with the top three algorithms in the challenge in

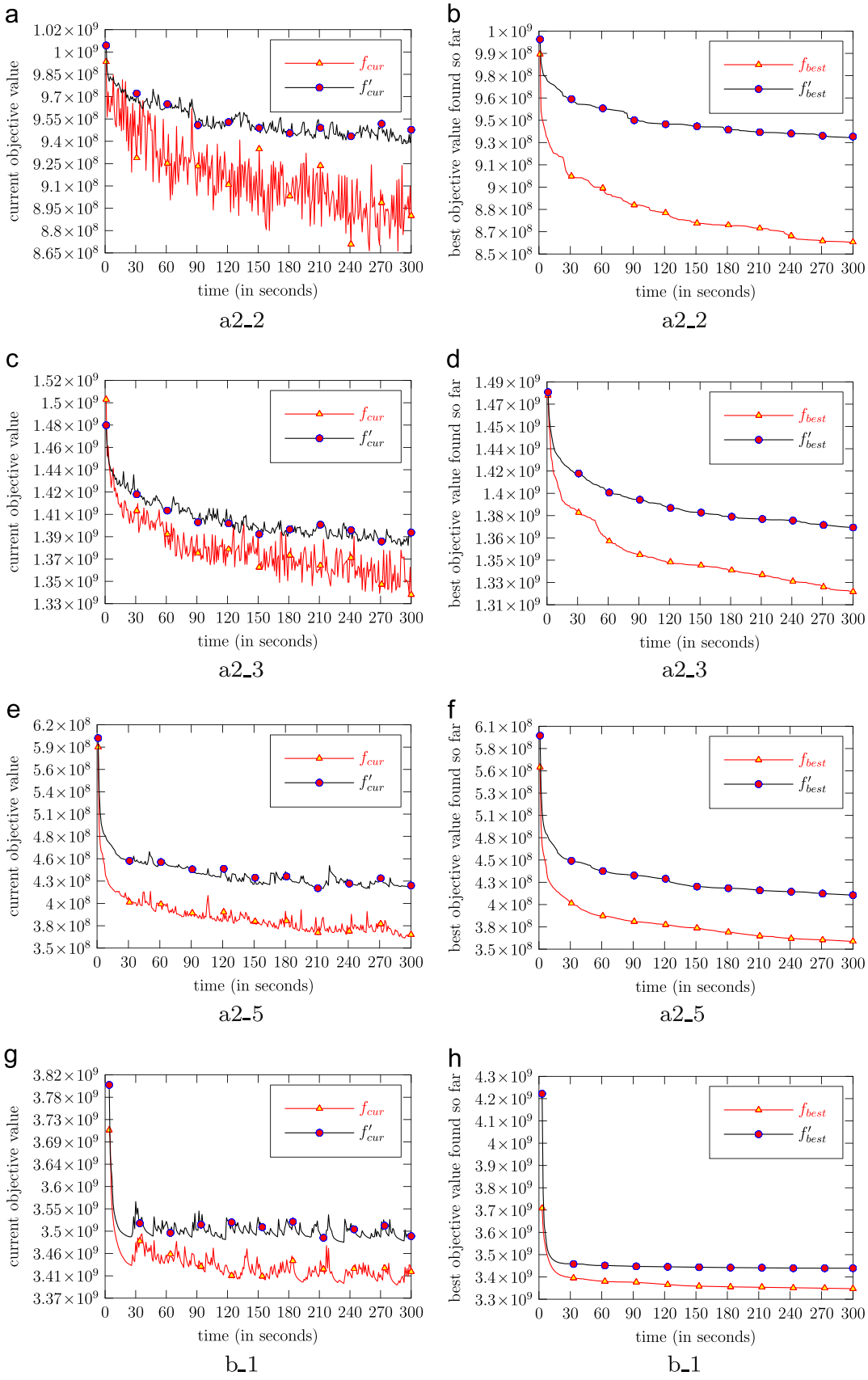

Fig. 3. Importance of infeasible moves. The curves show how the current objective value and the best objective value evolve with time, where f_{cur} and f_{best} represent the current and the best objective value obtained with applying infeasible moves, f'_{cur} and f'_{best} represent the respective results obtained without applying infeasible moves. (a) a2_2, (b) a2_2, (c) a2_3, (d) a2_3, (e) a2_5, (f) a2_5, (g) b_1, (h) b_1.

Table 8
Solution comparison between the algorithms with and without infeasible moves.

Inst.	case	0–20 (%)	20–40 (%)	40–60 (%)	60–80 (%)	80–100 (%)	Total (%)
a2_2	with	11.15	17.25	25.40	41.90	75.10	34.16
	without	10.85	16.10	26.40	41.20	73.95	33.70
a2_3	with	7.85	21.30	23.90	49.00	80.60	36.53
	without	5.85	19.70	25.95	48.50	81.60	36.32
a2_5	with	21.85	72.05	89.50	96.25	98.00	75.53
	without	14.10	63.05	87.05	95.60	97.25	71.41
b_1	with	9.81	17.75	23.85	37.43	69.32	31.63
	without	8.90	18.34	23.93	34.88	66.57	30.52
Average	with	12.67	32.09	40.66	56.15	80.76	44.46
	without	9.93	29.30	40.83	55.05	79.84	42.99

Table 9
Computational results under 300 s timeout condition.

Inst.	Init. cost	BKS	MNLS				
			f_{best}	f_{avr}	r (%)	t_{best}	c_v (%)
a1_1	49,528,750	44,306,501	44,306,501	44,306,501	0.00	0	0.00
a1_2	1,061,649,570	777,532,177	777,535,597	778,992,629	0.00	207	0.47
a1_3	583,662,270	583,005,717	583,005,717	583,005,718	0.00	126	0.00
a1_4	632,499,600	244,875,206	248,324,245	253,820,491	1.41	246	1.12
a1_5	782,189,690	727,578,309	727,578,309	727,578,327	0.00	186	0.00
a2_1	391,189,190	161	225	249	39.75	222	4.70
a2_2	1,876,768,120	720,671,548	793,641,799	855,269,586	10.13	289	2.43
a2_3	2,272,487,840	1,190,713,414	1,251,407,669	1,329,737,611	5.10	257	1.81
a2_4	3,223,516,130	1,680,368,578	1,680,744,868	1,681,321,215	0.02	219	0.02
a2_5	787,355,300	307,150,825	337,363,179	356,218,288	9.84	286	1.78
b_1	7,644,173,180	3,291,069,369	3,354,204,707	3,392,876,548	1.92	300	0.79
b_2	5,181,493,830	1,015,496,187	1,021,230,060	1,026,996,342	0.56	258	0.38
b_3	6,336,834,660	156,691,279	157,127,101	157,592,230	0.28	77	1.20
b_4	9,209,576,380	4,677,808,036	4,677,895,984	4,677,908,048	0.00	10	0.00
b_5	12,426,813,010	922,944,697	923,427,881	923,464,289	0.05	85	0.00
b_6	12,749,861,240	9,525,851,483	9,525,885,495	9,525,889,553	0.00	9	0.00
b_7	37,946,901,700	14,834,456,201	14,842,926,007	14,848,208,283	0.06	279	0.02
b_8	14,068,207,250	1,214,291,143	1,214,591,033	1,214,932,744	0.02	46	0.12
b_9	23,234,641,520	15,885,437,256	15,885,541,403	15,885,575,538	0.00	19	0.00
b_10	42,220,868,760	18,048,187,105	18,055,765,224	18,059,404,460	0.04	244	0.01
x_1	7,422,426,760	3,044,418,078	3,060,461,509	3,126,591,348	0.53	295	1.00
x_2	5,103,634,830	1,002,379,317	1,010,050,981	1,015,411,112	0.77	257	0.42
x_3	6,119,933,380	69,970	493,917	627,660	605.90	61	19.81
x_4	9,207,188,610	4,721,591,023	4,721,727,496	4,721,752,868	0.00	13	0.00
x_5	12,369,526,590	54,132	518,250	596,862	857.38	72	8.28
x_6	12,753,566,360	9,546,936,159	9,546,966,175	9,546,970,015	0.00	8	0.00
x_7	37,763,791,230	14,252,476,508	14,259,657,575	14,276,946,112	0.05	300	0.13
x_8	11,611,565,600	29,193	83,711	197,860	186.75	233	33.10
x_9	23,146,106,380	16,125,562,162	16,125,675,266	16,125,720,746	0.00	20	0.00
x_10	42,201,640,770	17,815,989,054	17,824,568,855	17,829,673,543	0.05	254	0.01

Table 11. Column 2 gives the best solution cost among all the competitors. The first three places are labeled with S41 [6], S38 [4,5] and J12 [11], respectively. For each method, we give its results (column cost) and the differential ratios between our best results obtained under 300 s timeout condition and its results (column $r(\%)$), i.e., $\frac{f_{best}-\text{cost}}{\text{cost}} \times 100\%$. In the last two rows, we present the outcome statistics where row "better" ("worse") means the number of instances for which our results are better (worse) than those of the corresponding method.

From the comparison in Table 11, one finds that the performances of the four algorithms are quite close to each other in general. There does not exist an algorithm which can dominate other ones. Specifically, our MNLS algorithm obtains better results for 10 instances compared with S41, 10 instances with S38, and 11 instances with J12, respectively. This experiment demonstrates that our MNLS algorithm is comparable with the methods of the first three places in the challenge.

5.3. Comparison with the reference algorithms in the literature

In this section, we compare our computational results under 300 s timeout condition with that of the state-of-the-art reference algorithms in the literature. These reference algorithms include a variable neighborhood search by Gavranović et al. [6], a CP-based large neighborhood search by Mehta et al. [4], a large neighborhood search that uses a constraint program by Brandt et al. [7], a multi-start iterated local search by Masson et al. [8], a simulated annealing by Portal [9] and a restricted iterated local search by Lopes et al. [13]. Table 12 shows the detailed comparison with these reference algorithms. For each reference algorithm, the objective cost, the ratio and the outcome statistic as defined above are presented. Notice that several other references [5,11,12] have also studied the MRP problem. However, they either did not report specific best results or just presented their best results in terms of the relative percentage or gap. Therefore, the results of these references are not reported here.

Table 10
Computational results under 3600 s timeout condition.

Inst.	Init. cost	BKS	MNLS				
			f_{best}	f_{avr}	r (%)	t_{best}	c_v (%)
a1_1	49,528,750	44,306,501	44,306,501	44,306,501	0.00	0	0.00
a1_2	1,061,649,570	777,532,177	777,534,387	777,553,716	0.00	3226	0.01
a1_3	583,662,270	583,005,717	583,005,717	583,005,717	0.00	395	0.00
a1_4	632,499,600	244,875,206	246,182,157	249,455,645	0.53	3563	0.68
a1_5	782,189,690	727,578,309	727,578,309	727,578,311	0.00	246	0.00
a2_1	391,189,190	161	213	232	32.30	2909	3.82
a2_2	1,876,768,120	720,671,548	757,990,293	814,399,746	5.18	3271	3.54
a2_3	2,272,487,840	1,190,713,414	1,222,464,496	1,277,247,204	2.67	3593	2.30
a2_4	3,223,516,130	1,680,368,578	1,680,517,169	1,680,663,286	0.01	3510	0.01
a2_5	787,355,300	307,150,825	326,479,641	333,750,995	6.29	3598	1.12
b_1	7,644,173,180	3,291,069,369	3,313,424,804	3,344,598,430	0.68	3554	0.53
b_2	5,181,493,830	1,015,496,187	1,017,964,474	1,019,344,611	0.24	2366	0.09
b_3	6,336,834,660	156,691,279	157,135,058	157,302,740	0.28	67	0.15
b_4	9,209,576,380	4,677,808,036	4,677,902,412	4,677,908,247	0.00	10	0.00
b_5	12,426,813,010	922,944,697	923,438,815	923,474,009	0.05	88	0.00
b_6	12,749,861,240	9,525,851,483	9,525,860,169	9,525,861,964	0.00	3435	0.00
b_7	37,946,901,700	14,834,456,201	14,837,141,814	14,838,695,505	0.02	3458	0.01
b_8	14,068,207,250	1,214,291,143	1,214,598,336	1,214,652,754	0.03	45	0.02
b_9	23,234,641,520	15,885,437,256	15,885,548,254	15,885,587,587	0.00	20	0.00
b_10	42,220,868,760	18,048,187,105	18,050,101,848	18,052,753,379	0.01	3218	0.01
x_1	7,422,426,760	3,044,418,078	3,034,666,101	3,065,392,246	−0.32	3019	0.52
x_2	5,103,634,830	1,002,379,317	1,006,826,798	1,009,023,714	0.44	2942	0.11
x_3	6,119,933,380	69,970	523,757	666,128	648.55	79	20.99
x_4	9,207,188,610	4,721,591,023	4,721,733,778	4,721,752,780	0.00	14	0.00
x_5	12,369,526,590	54,132	522,890	606,179	865.95	76	7.80
x_6	12,753,566,360	9,546,936,159	9,546,942,243	9,546,943,386	0.00	2457	0.00
x_7	37,763,791,230	14,252,476,508	14,255,705,702	14,256,851,153	0.02	3300	0.01
x_8	11,611,565,600	29,193	62,359	72,534	113.61	3359	10.52
x_9	23,146,106,380	16,125,562,162	16,125,684,506	16,125,712,997	0.00	18	0.00
x_10	42,201,640,770	17,815,989,054	17,818,594,096	17,820,917,288	0.01	3396	0.01

Table 11
Comparison with the results of the top three places in the challenge.

Inst.	Challenge	f_{best}	S41 [6]		S38 [4,5]		J12 [11]	
			cost	r (%)	cost	r (%)	cost	r (%)
b_1	3,339,186,879	3,354,204,707	3,370,468,979	−0.48	3,339,186,879	0.45	3,409,213,651	−1.61
b_2	1,015,553,800	1,021,230,060	1,015,553,800	0.56	1,023,619,894	−0.23	1,015,611,417	0.55
b_3	156,835,787	157,127,101	157,630,828	−0.32	157,435,411	−0.20	156,928,092	0.13
b_4	4,677,823,040	4,677,895,984	4,677,980,888	0.00	4,677,828,598	0.00	4,677,996,750	0.00
b_5	923,092,380	923,427,881	924,247,250	−0.09	923,589,955	−0.02	923,242,548	0.02
b_6	9,525,857,752	9,525,885,495	9,525,910,853	0.00	9,525,868,827	0.00	9,525,923,814	0.00
b_7	14,835,149,752	14,842,926,007	14,835,149,752	0.05	14,839,331,493	0.02	14,843,891,563	−0.01
b_8	1,214,458,817	1,214,591,033	1,214,581,898	0.00	1,214,564,460	0.00	1,222,308,107	−0.63
b_9	15,885,486,698	15,885,541,403	15,885,641,868	0.00	15,885,755,054	0.00	15,885,703,848	0.00
b_10	18,048,515,118	18,055,765,224	18,048,515,118	0.04	18,054,660,218	0.01	18,050,876,811	0.03
x_1	3,100,852,728	3,060,461,509	3,100,852,728	−1.30	3,104,898,127	−1.43	3,148,704,072	−2.80
x_2	1,002,502,119	1,010,050,981	1,003,467,532	0.66	1,014,840,104	−0.47	1,003,434,915	0.66
x_3	211,656	493,917	388,271	27.21	709,591	−30.39	211,656	133.36
x_4	4,721,629,497	4,721,727,496	4,721,857,756	0.00	4,721,629,497	0.00	4,721,902,699	0.00
x_5	93,823	518,250	257,261	101.45	93,823	452.37	146,884	252.83
x_6	9,546,941,232	9,546,966,175	9,546,971,975	0.00	9,546,951,163	0.00	9,546,985,251	0.00
x_7	14,253,273,178	14,259,657,575	14,255,223,202	0.03	14,290,228,022	−0.21	14,265,176,281	−0.04
x_8	42,674	83,711	157,441	−46.83	63,930	30.94	71,794	16.60
x_9	16,125,612,590	16,125,675,266	16,125,771,008	0.00	16,125,904,195	0.00	16,125,856,044	0.00
x_10	17,816,514,161	17,824,568,855	17,816,514,161	0.05	17,826,524,428	−0.01	17,822,032,910	0.01
better			10		10		11	
worse			10		10		9	

From Table 12, one finds that the best results obtained by our MNLS algorithm are competitive with respect to those of these 6 reference algorithms. Specifically, our MNLS algorithm can obtain better results for 8 instances compared with those of Gavranović et al., 10 instances compared with those of Mehta et al., 25 instances compared with those of Brandt et al., 28 instances compared with those of Masson et al., 10 instances compared with those of Portal and 13 instances compared with those of Lopes et al. Our results are the best among all the reference algorithms for 8 instances, and match the best result for instance a1_1.

Table 12
Comparison with the results in the literature.

Inst.	f_{best}	Gavranović [6]		Mehta [4]		Brandt [7]		Masson [8]		Portal [9]		Lopes [13]	
		cost	r (%)	cost	r (%)	cost	r (%)	cost	r (%)	cost	r (%)	cost	r (%)
a1_1	44,306,501	44,306,501	0.00	44,306,501	0.00	44,306,575	0.00	44,306,501	0.00	44,306,935	0.00	–	–
a1_2	777,535,597	777,536,907	0.00	778,654,204	−0.14	788,074,333	−1.34	780,499,081	−0.38	777,533,311	0.00	–	–
a1_3	583,005,717	583,005,818	0.00	583,005,829	0.00	583,006,204	0.00	583,006,015	0.00	583,009,439	0.00	-	-
a1_4	248,324,245	251,524,763	−1.27	251,189,168	−1.14	278,114,660	−10.71	258,024,574	−3.76	260,693,258	−4.74	-	-
a1_5	727,578,309	727,578,310	0.00	727,578,311	0.00	727,578,362	0.00	727,578,412	0.00	727,578,311	0.00	-	-
a2_1	225	199	13.07	196	14.80	1,869,113	−99.99	167	34.73	222	1.35	-	-
a2_2	793,641,799	720,671,548	10.13	803,092,387	−1.18	858,367,123	−7.54	970,536,821	−18.23	877,905,951	−9.60	-	-
a2_3	1,251,407,669	1,190,713,414	5.10	1,302,235,463	−3.90	1,349,029,713	−7.24	1,452,810,819	−13.86	1,380,612,398	−9.36	-	-
a2_4	1,680,744,868	1,680,615,425	0.01	1,683,530,845	−0.17	1,689,370,535	−0.51	1,695,897,404	−0.89	1,680,587,608	0.01	-	-
a2_5	337,363,179	309,714,522	8.93	331,901,091	1.65	385,272,187	−12.44	412,613,505	−18.24	310,243,809	8.74	-	-
b_1	3,354,204,707	3,307,124,603	1.42	3,337,329,571	0.51	3,421,883,971	−1.98	3,516,215,073	−4.61	3,455,971,935	−2.94	3,511,150,815	−4.47
b_2	1,021,230,060	1,015,517,386	0.56	1,022,043,596	−0.08	1,031,415,191	−0.99	1,027,393,159	−0.60	1,015,763,028	0.54	1,017,134,891	0.40
b_3	157,127,101	156,978,411	0.09	157,273,705	−0.09	163,547,097	−3.93	158,027,548	−0.57	215,060,097	−26.94	161,557,602	−2.74
b_4	4,677,895,984	4,677,961,007	0.00	4,677,817,475	0.00	4,677,869,484	0.00	4,677,940,074	0.00	4,677,985,338	0.00	4,677,999,380	0.00
b_5	923,427,881	923,610,156	−0.02	923,335,604	0.01	940,312,257	−1.80	923,857,499	−0.05	923,299,310	0.01	923,732,659	−0.03
b_6	9,525,885,495	9,525,900,218	0.00	9,525,867,169	0.00	9,525,862,018	0.00	9,525,913,044	0.00	9,525,861,951	0.00	9,525,937,918	0.00
b_7	14,842,926,007	14,835,031,813	0.05	14,838,521,000	0.03	14,868,550,671	−0.17	15,244,960,848	−2.64	14,836,763,304	0.04	14,835,597,627	0.05
b_8	1,214,591,033	1,214,416,705	0.01	1,214,524,845	0.01	1,219,238,781	−0.38	1,214,930,327	−0.03	1,214,563,084	0.00	1,214,900,909	−0.03
b_9	15,885,541,403	15,885,548,612	0.00	15,885,734,072	0.00	15,887,269,801	−0.01	15,885,617,841	0.00	15,886,083,835	0.00	15,885,632,605	0.00
b_10	18,055,765,224	18,048,499,616	0.04	18,049,556,324	0.03	18,092,883,448	−0.21	18,093,202,104	−0.21	18,049,089,128	0.04	18,052,239,907	0.02
x_1	3,060,461,509	-	-	-	-	3,119,249,147	−1.88	3,209,874,890	−4.65	-	-	3,341,920,446	−8.42
x_2	1,010,050,981	-	-	-	-	1,018,164,308	−0.80	1,018,646,825	−0.84	-	-	1,008,340,365	0.17
x_3	493,917	-	-	-	-	4,784,450	−89.68	1,965,401	−74.87	-	-	1,359,493	−63.67
x_4	4,721,727,496	-	-	-	-	4,721,702,912	0.00	4,721,786,173	0.00	-	-	4,721,833,040	0.00
x_5	518,250	-	-	-	-	391,923	32.23	615,277	−15.77	-	-	385,150	34.56
x_6	9,546,966,175	-	-	-	-	9,546,945,537	0.00	9,546,992,887	0.00	-	-	9,547,002,140	0.00
x_7	14,259,657,575	-	-	-	-	14,330,862,773	−0.50	14,701,830,252	−3.01	-	-	14,253,835,332	0.04
x_8	83,711	-	-	-	-	98,054	−14.63	309,080	−72.92	-	-	96,936	−13.64
x_9	16,125,675,266	-	-	-	-	16,128,419,926	−0.02	16,125,753,242	0.00	-	-	16,125,780,091	0.00
x_10	17,824,568,855	-	-	-	-	17,861,616,489	−0.21	17,867,789,754	−0.24	-	-	17,819,116,915	0.03
better		8/20		10/20		25/30		28/30		10/20		13/20	
equal		1/20		1/20		0/30		1/30		0/20		0/20	
worse		11/20		9/20		5/30		1/30		10/20		7/20	

6. Conclusion

In this paper, we have studied the machine reassignment problem which is the topic of the Google ROADEF/EURO Challenge 2012. In addition to providing a mathematical formulation of the problem, we have presented an effective multi-neighborhood local search based metaheuristic algorithm for solving the MRP. Our proposed algorithm integrates a number of important features, including three primary and one auxiliary neighborhood structures, a neighborhood partition mechanism and a dynamic perturbation operator. Our local search procedure is based on the three augmented primary neighborhoods with the auxiliary neighborhood, which allows our algorithm to perform infeasible moves and bring more diversification into the search. The neighborhood partition search mechanism is beneficial to speed up the search and the perturbation operator helps the algorithm to escape from local optima.

Several key components of our proposed MNLS algorithm have been investigated. First of all, experiments have been conducted to demonstrate the influence of proper neighborhood partition on the performance of our algorithm. Secondly, we have carried out experiments to confirm the significance of the three primary neighborhood structures. Finally, we have shown the importance of applying infeasible moves.

Furthermore, the performance of our MNLS algorithm has been extensively assessed on benchmark instances under the ROADEF/EURO Challenge 2012 timeout condition. The comparisons with the best results in the challenge and the literature have demonstrated the effectiveness of our MNLS algorithm.

Finally, it is worthy to notice that many ideas of the proposed algorithm, e.g., neighborhood partition mechanism, the technique to allow infeasible moves, are independent of the MRP problem, and it would be valuable to examine their applications to other combinatorial optimization and constraint satisfaction problems, such as the generalized assignment problem. The MRP is a challenging problem in load balance of cloud computing. The study in this work motivates us to further develop a robust and fast method to meet various real-world requirements.

Acknowledgments

We thank the anonymous referees for their valuable comments which have helped us to improve our paper significantly. This work was partially supported by the National Nature Science Foundation of China (Grant nos. 61370183 and 61100144) and the Program for New Century Excellent Talents in University (NCET 2013).

References

[1] Armbrust M, Fox A, Griffith R, Joseph A, Katz R, Konwinski A, et al. A view of cloud computing. Commun ACM 2010;53(4):50–8.
[2] Beloglazov A, Abawajy J, Buyya R. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. Future Gener Comput Syst 2012;28(5):755–68.
[3] ROADEF/EURO Challenge 2012: Machine reassignment. ⟨http://challenge.roadef.org/2012/en/⟩.
[4] Mehta D, O'Sullivan B, Simonis H. Comparing solution methods for the machine reassignment problem. In: Milano M, editor. Principles and practice of constraint programming, lecture notes in computer science. Heidelberg: Springer; 2012. p. 782–97.
[5] Malitsky Y, Mehta D, O'Sullivan B, Simonis H. Tuning parameters of large neighborhood search for the machine reassignment problem. In: Gomes C, Sellmann M, editors. Integration of AI and OR techniques in constraint programming for combinatorial optimization problems, lecture notes in computer science, vol. 7874. Heidelberg: Springer; 2013. p. 176–92.
[6] Gavranović H, Buljubašić M, Demirović E. Variable neighborhood search for google machine reassignment problem. Electron Notes Discret Math 2012;39:209–16.
[7] Brandt F, Speck J, Vöker M. Constraint-based large neighborhood search for machine reassignment. Ann Oper Res 2014. ⟨http://dx.doi.org/10.1007/s10479-014-1772-6⟩.
[8] Masson R, Vidal T, Michallet J, Penna PHV, Petrucci V, Subramanian A, et al. An iterated local search heuristic for multi-capacity bin packing and machine reassignment problems. Expert Syst Appl 2013;40(13):5266–75.
[9] Portal G. An algorithmic study of the machine reassignment problem. Technical report, Federal University of Rio Grande do Sul;2012.
[10] Kirkpatrick S, Gelatt C, Vecchi M. Optimization by simulated annealing. Science 1983;220(4598):671–80.
[11] Jaśkowski W, Szubert M, Gawron P. A hybrid MIP-based large neighborhood search heuristic for solving the machine reassignment problem. Ann Oper Res 2015. ⟨http://dx.doi.org/10.1007/s10479-014-1780-6⟩.
[12] Portal GM, Ritt M, Borba LM, Buriol LS. Simulated annealing for the machine reassignment problem. Ann Oper Res 2015. ⟨http://dx.doi.org/10.1007/s10479-014-1771-7⟩.
[13] Lopes R, Morais VW, Noronha TF, Souza VA. Heuristics and matheuristics for a real-life machine reassignment problem. Int Trans Oper Res 2015;22(1):77–95.
[14] Pentico DW. Assignment problems: a golden anniversary survey. Eur J Oper Res 2007;176(2):774–93.
[15] Horn G, Oommen BJ. Solving multiconstraint assignment problems using learning automata. IEEE Trans Syst Man Cybern, B 2010;40(1):6–18.
[16] Sun J, Zhang Q, Yao X. Meta-heuristic combining prior online and offline information for the quadratic assignment problem. IEEE Trans Cybern 2014;44(3):429–44.
[17] Cattrysse D, Wassenhove L. A survey of algorithms for the generalized assignment problem. Eur J Oper Res 1992;60(3):260–72.
[18] Nauss RM. The generalized assignment problem. In: Karloff JK, editor. Integer programming. Boca Raton: CRC Press; 2006. p. 39–55.
[19] Fisher M, Jaikumar R. A generalized assignment heuristic for vehicle routing. Networks 1981;11(2):109–24.
[20] Baldacci R, Mingozzi A. A unified exact method for solving different classes of vehicle routing problems. Math Program 2009;120(2):347–80.
[21] Baldacci R, Mingozzi A, Roberti R. New route relaxation and pricing strategies for the vehicle routing problem. Oper Res 2011;59(5):1269–83.
[22] Ross G, Soland R. Modeling facility location problems as generalized assignment problems. Manag Sci 1977;24(3):345–57.
[23] Shen Z-JM, Zhan RL, Zhang J. The reliable facility location problem: formulations, heuristics, and approximation algorithms. INFORMS J Comput 2011;23(3):470–82.
[24] Tang L, Jiang W, Saharidis GKD. An improved benders decomposition algorithm for the logistics facility location problem with capacity expansions. Ann Oper Res 2013;210(1):165–90.
[25] Amini M, Racer M. A hybrid heuristic for the generalized assignment problem. Eur J Oper Res 1995;87(2):343–8.
[26] Glover F, Laguna M. Tabu search. Boston: Kluwer Academic; 1997.
[27] Osman I. Heuristics for the generalised assignment problem: simulated annealing and tabu search approaches. Oper Res Spektrum 1995;17(4):211–25.
[28] Holland J. Adaptation in natural and artificial systems. Michigan: University of Michigan Press; 1975.
[29] Chu P, Beasley J. A genetic algorithm for the generalised assignment problem. Comput Oper Res 1997;24(1):17–23.
[30] Yagiura M, Yamaguchi T, Ibaraki T. A variable depth search algorithm with branching search for the generalized assignment problem. Optim Methods Softw 1998;10(2):419–41.
[31] Yagiura M, Yamaguchi T, Ibaraki T. A variable depth search algorithm for the generalized assignment problem. In: Voß S, Martello S, Osman I, Roucairol C, editors. Meta-heuristics: advances and trends in local search paradigms for optimization. Boston: Kluwer Academic; 1999. p. 459–71.
[32] Lau T, Tsang E. The guided genetic algorithm and its application to the generalized assignment problem. In: Proceedings of tenth IEEE int'l conference tools with artificial intelligence; 1998. p. 336–43.
[33] Feo T, Resende M. Greedy randomized adaptive search procedures. J Global Optim 1995;6(2):109–33.
[34] Stützle T, Hoos H. MAX-MIN ant system. Future Gener Comput Syst 2000;16(8):889–914.
[35] Lourenço H, Serra D. Adaptive search heuristics for the generalized assignment problem. Mathw Soft Comput 2002;9(3):209–34.
[36] Yagiura M, Ibaraki T, Glover F. An ejection chain approach for the generalized assignment problem. INFORMS J Comput 2004;16(2):133–51.
[37] Yagiura M, Ibaraki T, Glover F. A path relinking approach with ejection chains for the generalized assignment problem. Eur J Oper Res 2006;169(2):548–69.
[38] Tasgetiren M, Suganthan P, Chua T, Al-Hajri A. Differential evolution algorithms for the generalized assignment problem. In: Proceedings of IEEE congress evolutionary computation (CEC'09); 2009. p. 2606–13.
[39] Özbakir L, Baykasoğlu A, Tapkan P. Bees algorithm for generalized assignment problem. Appl Math Comput 2010;215(11):3782–95.
[40] Ross G, Soland R. A branch and bound algorithm for the generalized assignment problem. Math Program 1975;8(1):91–103.
[41] Savelsbergh M. A branch-and-price algorithm for the generalized assignment problem. Oper Res 1997;45(6):831–41.
[42] Haddadi S, Ouzia H. Effective algorithm and heuristic for the generalized assignment problem. Eur J Oper Res 2004;153(1):184–90.
[43] Woodcock AJ, Wilson JM. A hybrid tabu search/branch & bound approach to solving the generalized assignment problem. Eur J Oper Res 2010;207(2):566–578.