A cardinality constrained iterated local search for the virtual machine placement problem

Qing Zhou[a], Yuru Li[a], Jin-Kao Hao[b], Qinghua Wu[c], Yuning Chen[d,*]

[a]*School of Business Administration, Northeastern University, 195 Chuangxin Road, Shenyang 110169, China, email: zhouqing@mail.neu.edu.cn; liyuru0325@163.com*
[b]*LERIA, Université d'Angers, 2 bd Lavoisier, 49045 Angers Cedex 01, France, email: jin-kao.hao@univ-angers.fr*
[c]*School of Management, Huazhong University of Science and Technology, No. 1037, Luoyu Road, Wuhan, China, email: qinghuawu1005@gmail.com*
[d]*College of Systems Engineering, National University of Defense Technology, Changsha, 410073, China, email: yuning_ chen@nudt.edu.cn*

Computers & Operations Research 184: 107222, 2025
https://doi.org/10.1016/j.cor.2025.107222

Abstract

The virtual machine placement (VMP) problem is a critical task in the field of cloud computing. The assignment of virtual machines to physical machines affects the quality of cloud services and running cost. Given a set of physical machines with certain capacities and a set of virtual machines with requirements, VMP aims to allocate each virtual machine to a capacity constrained physical machine in such a way that the total number of the physical machines used is minimized while their usage does not exceed the capacity. In this study, a cardinality constrained iterated local search algorithm is proposed to solve the VMP problem by transforming VMP into a sequence of cardinality-constrained problems, where each problem involves a fixed number k of physical machines. The algorithm uses the tabu search procedure for solution improvement, which exploits two new neighborhoods based on dedicated evaluation functions for neighboring solution selection. In addition, it uses a simple perturbation strategy to prevent the algorithm from search stagnation. Numerical results show that the proposed algorithm is highly competitive in both solution quality and computational efficiency, compared to several state-of-the-art algorithms on 18 subsets of 1800 widely used benchmark instances. Specifically, the algorithm reports the best results in terms of the average objective values on 17 out of 18 instance subsets with

*Corresponding author.

a short run time of 5 seconds. Importantly, using the lower bounds, it proves for the first time the optimality of solutions for 1390 instances. We study the impact of the key components of the algorithm on its performance.

Keywords: Metaheuristics; virtual machine placement; local search; combinatorial optimization.

---

1. Introduction

Cloud computing is a powerful computing paradigm in which various services such as software and infrastructures are provided by cloud platforms to the remote users [14]. Virtualization technology enables the creation of virtual machines (VMs) for the user application with the required resources, which are then assigned to physical machines (PMs) to run the application in the data center. However, the data centers deployed in cloud platforms to provide cloud services consume large amounts of energy. It has been shown that data centers contribute about 1% of the world's electricity consumption in 2020, and the energy consumed by the world's data centers will continue to grow rapidly in the coming years [26]. An active but idle PM consumes approximately $50\% \sim 70\%$ as much energy as a fully utilized PM [21]. Therefore, consolidation of VMs on PMs is becoming increasingly important to reduce the energy consumption [37]. This leads to the virtual machine placement (VMP) problem, which aims to achieve an optimal allocation of VMs to PMs. Moreover, the scale of this problem is relatively large. In 2011, Google released a one-month trace of its cloud workload, which includes approximately 650,000 tasks assigned to 12,000 servers within a data center [15]. VMP is computationally challenging as it has been shown to be NP-hard [32]. The mathematical model of VMP is presented in Section 3.1.

Due to the importance of VMP in cloud resource management, a number of optimization algorithms have been proposed in the literature as reviewed in Section 2. These existing VMP approaches have contributed to finding satisfactory solutions for a number of benchmark instances of VMP. However, their performance varies depending on the test graphs, and they face the challenge of consistently generating high-quality solutions for various graphs, e.g., with a percentage gap between the yielded solution and the lower bound generally exceeding 2%. This work aims thus to advance the state-of-the-art for effectively tackling VMP by proposing a very competitive heuristic method, which is able to find high quality solutions in a short computing

time on the commonly used benchmark instances. The main contributions of this work are summarized as follows.

From an algorithmic design perspective, the proposed cardinality constrained iterated local search (CCILS) algorithm solves the initial VMP problem by solving a series of cardinality constrained problems. Specifically, CCILS seeks a solution of fixed size k by exploring a constrained space where exactly k physical machines are used. For a given candidate solution, CCILS uses a tabu search procedure for solution improvement by applying two new neighborhoods induced by the constrained exchange (denoted as $Cons\_Ex$) and $Migrate$ operators, based on dedicated evaluation functions for neighboring solution selection. Particularly, the constrained exchange operator tries to swap virtual machines between non-overloaded physical machines. To prevent the algorithm from stagnation, a simple perturbation strategy is used. After finding a feasible solution, i.e., obtaining a feasible placement of all the VMs on k PMs, k is decreased by 1, and a new solution of fixed size k is sought. This strategy differs from the conventional VMP methods that primarily focus on directly constructing feasible solutions. By exploring the solution space from infeasible regions, the proposed CCILS algorithm potentially uncovers solution space that conventional approaches may overlook. Meanwhile, compared to VMP algorithms that are mainly based on the population optimization framework, the proposed CCILS algorithm is based on the local search scheme that does not require the population management component. Additional experiments have been conducted to isolate and demonstrate the contribution of the proposed components, including the constrained exchange operator and the dedicated evaluation functions in Section 5, and the experimental results support their positive impact on the overall performance of the algorithm.

From the point of view of computational results, the experimental evaluations demonstrate the high competitiveness of the proposed algorithm compared to several state-of-the-art algorithms on the 18 subsets of 1800 widely used benchmark instances. Specifically, CCILS gives the best results in terms of the average objective values in 17 out of 18 subsets of instances with a time limit of 5 seconds for each execution, while the state-of-the-art methods produce the best result at most on 1 instance subset. Furthermore, using the lower bounds, it identifies for the first time 1390 optimal solutions out of the total of 1800 instances. The source code of the algorithm will be made publicly available, which can help researchers and practitioners to better solve various practical problems that can be formulated as VMP.

We further consider the time dimension into VMP to broaden the practical applicability of the proposed method.

The rest of the paper is organized as follows. Section 2 reviews existing related works. Section 3 presents the problem and its mathematical formulation, as well as the main components of the proposed algorithm. Computational results and comparisons with the state-of-the-art methods are given in Section 4. Section 5 examines the effect of important components of CCILS. Conclusions and perspectives are discussed in the last section.

2. Literature review

There are a variety of VMP scenarios have been extensively studied, such as energy-aware VMP, network-aware VMP and resource-aware VMP. As the energy consumption of large data centers becomes increasingly prominent, the energy-aware VMP problem focusing on minimizing the energy consumption has been widely concerned [6, 23]. The network-aware VMP problem concentrates on the overall network traffic by minimizing the average path length between VMs [8, 18]. The resource-aware VMP problem focuses on maximizing the utilization of server resources, including storage resources, computing resources and memory resources, etc [34]. Besides, multi-objective VMP problems have also been widely studied [9, 37]. Abbasi-khazaei and Rezvani [1] presented a modified memetic algorithm to jointly minimize energy costs and scheduling costs. Below, we review the existing methods for solving VMP reported in the literature.

In [24], Liu et al. proposed an ant colony optimization algorithm with order exchange and migration local search techniques, termed as OEMACS to tackle VMP. The allocation of VMs is undertaken by artificial ants based on global search information. OEMACS distributes pheromones between VM pairs on the same server that represents connections between VMs, and records accumulated historical experience with pheromones. Local search is integrated into OEMACS to try to fix infeasible solutions. The experimental results show that OEMACS generally outperforms the current best performing algorithms [33, 35], especially on VMP instances with bottleneck resources. Despite its effectiveness on small- and medium-sized instances, OEMACS may lead to significant computational overhead when applied to large-scale problems, owing to its experimentally observed time complexity of approximately $O(n^4)$ [29].

In [2], Abohamama and Hamouda presented the improved genetic algorithm (denoted as GA1) for VMP. The proposed algorithm combines a modified permutation based genetic algorithm and a resource-aware best-fit strategy. GA1 uses a fitness function that prioritizes low power usage, and the comparison experiments show that GA1 performs competitively in terms of solution quality against several other (simple) algorithms. The evaluation, crossover, and mutation operations in each generation of GA1 are computationally intensive, with an approximate time complexity of $O(n^3)$. Moreover, although GA1 emphasizes global exploration, it lacks fine-grained local search capabilities. However, incorporating a local search procedure could further increase the algorithm's time efficiency, which could become an even more serious bottleneck, particularly for large-scale problems.

In 2022, Peake et al. [29] developed an improved algorithm for VMP, based on parallel ant colony optimization (PACO), which significantly improves the computational efficiency through parallelization techniques (such as OpenMP and AVX2 instruction sets). The computational results on three sets of 1800 benchmark instances show that PACO is several orders of magnitude faster than the state-of-the-art methods, including OEMACS, GA1, and an adaption of GA1 (denoted as GA2), while yielding comparable results. The main advantage of PACO lies in its superior time efficiency compared to the reference algorithms, as it exhibits a time complexity of $O(n^2)$. However, the algorithm is more challenging to implement, particularly its acceleration strategies, which rely on a specialized parallel framework and modern processor technologies. This could pose certain difficulties for those attempting to apply the method to other related VMP problems.

According to the numerical results reported in VMP literature, OEMACS [24], GA1 [2], GA2 [2] and PACO [29] are the current best performing methods for VMP. However, there is still significant room for improvement in solution quality and computational efficiency. In fact, existing VMP algorithms do not perform consistently well on the benchmark instances, and no single method is able to achieve all best-known results on the benchmark instances. In addition, some methods, such as OEMACS, GA1, GA2 take a long time to achieve their results. Finally, one can observe that the existing VMP algorithms are mainly based on the population optimization framework, while the single solution based heuristics remain relatively limited. It is known that the iterated local search, one of the single solution based heuristic methods, is a powerful general framework for combinatorial optimization problems [25]. Until now, this approach is still

little studied for solving VMP. This work therefore aims to fill these gaps by investigating, for the first time, the potential of the iterated local search framework to address VMP. As shown by the computational results in Section 4, the proposed algorithm is indeed very effective.

Meanwhile, VMP can be considered as a variation of the one-dimensional bin packing problem (BPP) [5, 11, 12]. Given a set of items, each having an integer weight, and an unlimited number of identical bins of integer capacity, BPP seeks to pack all the items into the minimum number of bins such that the total weight packed in any bin does not exceed the capacity. The physical machines, which correspond to bins, have limited capacity on one or more dimensions, and the virtual machines corresponding to items are characterized by the resources that they consume on the physical machines. The main difference between VMP and BPP is that VMP considers multiple resource dimensions simultaneously, such as CPU and RAM, whereas BPP involves only a single-dimensional weight. For a comprehensive review of the bin packing problem, we refer the reader to [12].

Most existing VMP studies focus on minimizing the number of active servers or operational costs for a given set of VM requests. However, a significant variation of the problem involves incorporating the time dimension, where customers request VMs for specific durations. Only a few studies have considered this aspect in their models or developed algorithms specifically to address it. For example, the temporal extension of the bin packing problem, known as the temporal bin packing (TBP) problem has been considered (e.g., [5, 10, 11]), where the items should be packed in fixed start and end times within a given planning horizon. Nevertheless, the time dimension is a critical factor in practical applications. To broaden the scope and applicability of this work, we further consider incorporating the time dimension into VMP and extend the current solution to address this critical aspect.

3. Cardinality constrained iterated local search for VMP

3.1. Problem statement and mathematical formulation

We consider a set of virtual machines $V = \{V_i, i \in \{1, ..., N_{VM}\}\}$ with CPU requirements and RAM requirements R_i^{CPU}, R_i^{RAM} for each $V_i \in V$, and a set of physical machines $P = \{P_m, m \in \{1, ..., N_{PM}\}\}$ with CPU capacities and RAM capacities C_m^{CPU}, C_m^{RAM} for each $P_m \in P$. The main assumptions of VMP are as follows: (i) each virtual machine should be assigned to exactly

one of the physical machines; (ii) the CPU and RAM requirements of each virtual machine do not exceed the capacities of a physical machine. The VMP consists of allocating all the virtual machines to the physical machines such that the physical machines' capacities are not violated. The objective of VMP is to minimize the number of used physical machines. Let y_m be a binary variable taking the value of 1 if P_m is used, and 0 otherwise. Let x_{im} be a binary variable such that $x_{im} = 1$ if V_i is allocated to P_m, and $x_{im} = 0$ otherwise. VMP can then be formally defined as follows [24].

$$\textbf{Minimize} \quad \sum_{m=1}^{N_{PM}} y_m \tag{1}$$

$$\textbf{subject to} \quad \sum_{m=1}^{N_{PM}} x_{im} = 1, \forall i \in \{1, ..., N_{VM}\} \tag{2}$$

$$\sum_{i=1}^{N_{VM}} x_{im} R_i^{CPU} \leq C_m^{CPU} y_m, \forall m \in \{1, ..., N_{PM}\} \tag{3}$$

$$\sum_{i=1}^{N_{VM}} x_{im} R_i^{RAM} \leq C_m^{RAM} y_m, \forall m \in \{1, ..., N_{PM}\} \tag{4}$$

$$x_{im} \in \{0, 1\}, \quad \forall i \in \{1, ..., N_{VM}\}, \forall m \in \{1, ..., N_{PM}\} \tag{5}$$

$$y_m \in \{0, 1\}, \quad \forall m \in \{1, ..., N_{PM}\} \tag{6}$$

The objective function (1) is to minimize the number of physical machines used. Constraint (2) ensures that each virtual machine is assigned to one and only one of the physical machines. Constraints (3) and (4) ensure that each used physical machine satisfies the resource requirements of the virtual machines on it. Binary values for the variables x_{im} and y_m are imposed in constraints (5) and (6), respectively.

3.2. Main scheme

Iterated local search is a metaheuristic that focuses the search on a sequence of solutions returning by some underlying method, typically a local search heuristic. Iterated local search is a conceptually simple metaheuristic approach, but it has led to state-of-the-art algorithms for a variety of computationally difficult problems, such as job shop scheduling problem [4, 16, 28] and vehicle routing problem [7, 27, 30].

Algorithm 1: General approach of the cardinality constrained iterated local search for VMP

Input: I: a VMP instance, t: the cutoff time, lb: the lower bound of I
Output: The best found solution (upper bound) S^*

```
1  S_k, flag ← greedy_ini(I) /* Generate greedily an initial k-machine
     solution */
2  S* ← S_k /* Update the best found solution */
3  k ← k − 1 /* Decrease the value of k by 1 */
4  /* Solve a series of cardinality constrained problems */
5  while the cutoff time t is not reached && k >= lb do
6      S_k, flag ← ILS(S_{k+1}, z) /* Invoke ILS method to solve VMP_k
         problem, see Algorithm 2 */
7      if the VMP_k is successfully solved, i.e., flag == True then
8          S* ← S_k
9          k ← k − 1
10     z ← t − clock() /* Update the cutoff time z of ILS */
11 return S*
```

To solve VMP problem more efficiently, we adopt a cardinality constrained strategy to decompose VMP into a series of problems, where each problem VMP_k with a cardinality of k physical machines focused on finding a feasible solution using exactly k physical machines denoted as P^k. Such a feasible solution, called the k-machine solution, is obviously an upper bound of the optimal VMP solution. To solve the VMP_k problem, the algorithm uses a penalty approach to explore the constrained search space with the fixed k physical machines. Each time a k-machine solution is found, k is decremented by one and a new k-machine solution is sought. The process is repeated until no k-machine solution can be found or the lower bound of k is reached (see Section 4.1 for calculating lower bound lb for an instance). The last feasible k-machine solution is thus the best upper bound found by the algorithm. Therefore, the VMP problem comes down to the problem of finding a k-machine solution. Note that the search strategy of cardinality constrained (also called the k-fixed penalty strategy), has been successfully used for graph coloring [19], clique problems [36, 38], stable set problem [17], etc.

The proposed CCILS algorithm presented in Algorithm 1 follows the idea above. To quickly obtain the first k value and the initial k-machine solution,

Algorithm 2: Iterated local search for VMP_k

```
Input: S_{k+1}: a (k+1)-machine solution, z: the cutoff time
Output: k-machine solution if found
1 S ← Initial_Solution(S_{k+1}) /* Section 3.3, generate an initial
    solution using k physical machines */
2 while the cutoff time z is not reached do
3     S ← Tabu_Search(S) /* Section 3.4, repair the solution */
4     if S is a k-machine solution then
5         return S and report success
6     S ← Perturbation_Solution(S) /* Section 3.5, perturb the
        current solution */
7 Report failure
```

a simple greedy method is used (line 1). For each V_i ($i \in \{1, ..., N_{VM}\}$), it is assigned to the first available PM that satisfies its resource requirements by visiting PMs in alphabetical order (i.e., from 1 to N_{PM}). Note that in a heterogeneous environment with different PMs, the PMs with the largest capacities are assigned the smallest numbers, in other words, they are visited first. This process is efficient and can handle large instances without a significant increase in processing time. The best found solution S^* is then updated using the initial k-machine solution (line 2) and the value of k is decremented by 1 (line 3). After that, CCILS uses the iterated local search method (ILS, see Algorithm 2) to a series of cardinality constrained problems VMP_k with decreasing k values. If VMP_k is successfully solved, i.e., a k-machine solution S_k is found with the current k value (lines 7), S^* is updated with S_k (line 8) and k is decreased by 1 (line 9). The cutoff time z for ILS is finally updated (line 10). CCILS is repeated until the cutoff time t is reached or the lower bound lb is arrived.

Algorithm 2 shows the pseduo-code of the ILS algorithm. Starting from an initial solution S (usually S is infeasible) (line 1, Section 3.3), ILS enters the 'while' loop (lines 2-6) to find an improved feasible solution. At each loop, a tabu search procedure is used to iteratively improve S (line 3, Section 3.4). A dedicated perturbation procedure is applied when the tabu search process gets stuck in a local optimal trap (line 6, Section 3.5). If a feasible k-machine solution is found, ILS stops and returns the feasible solution (lines 4-5). If no feasible k-machine solution is found within the given cutoff time, ILS stops

and reports a failure.

3.3. Solution initialization

Each input solution S of the local search is created by the solution initialization procedure (denoted as SIP). Given a $(k+1)$-machine solution, SIP first releases all the VMs on the physical machine P_{k+1}. Every V_i located on P_{k+1} is reallocated to a randomly selected physical machine P_r ($r \in \{1, 2, ..., k\}$). Note that during this procedure, CPU and RAM capacity constraints can be violated. Any constraint violation will be corrected during the subsequent tabu search procedure.

3.4. Tabu search procedure

The tabu search (TS) procedure is used for local improvement, relying on the general tabu search principle [20]. TS is characterized by its new move operators, the dedicated evaluation functions and an advanced exploration strategy.

3.4.1. Constrained move operators and dedicated evaluation functions

To improve the given solution, TS iteratively transits from the current solution S to a neighboring solution. For this purpose, TS uses two new move operators, i.e., the constrained exchange operator (denoted as $Cons\_Ex(i, j; m, n)$) and $Migrate$ operator for a more efficient search. To evaluate each neighboring solution, two dedicated evaluation functions are applied.

The $Cons\_Ex(i, j; m, n)$ operator: Let V_i and V_j be two virtual machines located on two non-overloaded physical machines $P_m, P_n \in P^k$ respectively. The $Cons\_Ex(i, j; m, n)$ operator swaps V_i and V_j without violating the capacity constraints of P_m and P_n. Note that the exchange operators utilized by other VMP heuristics [24, 29] mainly considered to swap VMs between an overloaded PM and a non-overloaded PM. The main reason behind our $Cons\_Ex(i, j; m, n)$ operator that considers only the non-overloaded PMs is to improve the search efficiency. The effect of the constrained exchange operator is investigated in Section 5.1.

Given the current solution S, let $S \oplus Cons\_Ex(i, j; m, n)$ be the neighboring solution obtained by applying $Cons\_Ex(i, j; m, n)$ to S. Then the constrained exchange neighborhood N_1 induced by the $Cons\_Ex(i, j; m, n)$

is defined as follows:

$$N_1(S) = \{S^{'} : S^{'} = S \oplus Cons\_Ex(i,j;m,n), V_i \in P_m, V_j \in P_n, 1 \leq m,n \leq k, \\ m \neq n, U_m^{CPU} \leq C_m^{CPU}, U_m^{RAM} \leq C_m^{RAM}, U_n^{CPU} \leq C_n^{CPU}, U_n^{RAM} \leq C_n^{RAM}, \\ U_m^{CPU} - R_i^{CPU} + R_j^{CPU} \leq C_m^{CPU}, U_m^{RAM} - R_i^{RAM} + R_j^{RAM} \leq C_m^{RAM}, \\ U_n^{CPU} - R_j^{CPU} + R_i^{CPU} \leq C_n^{CPU}, U_n^{RAM} - R_j^{RAM} + R_i^{RAM} \leq C_n^{RAM}\} \quad (7)$$

where U_m^{CPU} and U_m^{RAM} (U_n^{CPU} and U_n^{RAM}) represent the total CPU and RAM requirements of all virtual machines on P_m (P_n).

For each neighboring solution $S^{'} \in N_1(S)$, we apply a meticulously designed evaluation function r to evaluate the fitness of S', which is defined as follows:

$$r(S^{'}) = (2 + o_c + b_c) \times L_m^{CPU} + (1 + o_m + b_m) \times L_m^{RAM} \quad (8)$$

where L_m^{CPU} and L_m^{RAM} are the residual CPU and RAM resources of P_m after exchanging $V_i \in P_m$ and $V_j \in P_n$, respectively. Binary variables b_c and b_m are included in the function r to mark up the bottleneck resources of the instance. If CPU (RAM) is the bottleneck resource of the instance, $b_c = 1$ ($b_m = 1$), and $b_c = 0$ ($b_m = 0$) otherwise. Note that, the bottleneck resources of instances of Set B and Set C are CPU and RAM, respectively, while Set A instances have no bottleneck resource (See Section 4.1). The function r includes two other binary variables o_c and o_m whose values are related to the current solution S. By visiting PMs in alphabetical order (i.e., from 1 to k) and identifying the first overloaded PM $P_j \in P^k$, if its CPU (RAM) capacity is violated, then $o_c = 1$ ($o_m = 1$), and $o_c = 0$ ($o_m = 0$) otherwise. In this way, the remaining CPU and RAM resources of P_m after a $Cons\_Ex(i,j;m,n)$ move will be more suitable for accommodating VMs with different CPU and RAM requirements for the subsequent $Migrate$ neighborhood search. The natural numbers 1 and 2 in r are used to balance the differences of the CPU and RAM requirements of VMs as well as CPU and RAM capacities of PMs for the used instances.

The $Migrate(i,m,n)$ operator: Let V_i be a virtual machine located on a randomly selected overloaded physical machine $P_m \in P^k$. The $Migrate(i,m,n)$ operator displaces V_i from P_m to a non-overloaded physical machine $P_n \in P^k$ such that the capacity constraint of P_n is satisfied. The search efficiency of TS in exploring the $Migrate$ neighborhood is enhanced by constraining V_i from a randomly selected overloaded PM rather than

considering all overloaded PMs. The $Migrate$ neighborhood N_2 induced by the $Migrate(i, m, n)$ is defined as follows:

$$\begin{aligned} N_2(S) = \{S^{'} : S^{'} = S \oplus Migrate(i, m, n), V_i \in P_m, 1 \leq m, n \leq k, \\ m = rand(k), U_m^{CPU} > C_m^{CPU} \vee U_m^{RAM} > C_m^{RAM}, \\ U_n^{CPU} + R_i^{CPU} \leq C_n^{CPU}, U_n^{RAM} + R_i^{RAM} \leq C_n^{RAM}\} \end{aligned} \tag{9}$$

where the function $rand(k)$ returns an integer between 1 and k.

To evaluate each neighboring solution $S' \in N_2(S)$, a dedicated evaluation function d is defined by:

$$d(S^{'}) = (1 + o_c) \times R_i^{CPU} + (1 + o_m) \times R_i^{RAM} \tag{10}$$

A larger value of d indicates that the selected virtual machine V_i of P_m has a relatively greater demand, thus alleviating the overload degree of P_m more efficiently.

Obviously, for an initial solution using k PMs, the $Cons\_Ex(i, j; m, n)$ operator, which only considers non-overloaded PMs cannot find a k-machine solution. However, we can use it to adjust the placement of VMs and free up a PM with relatively large residual resources so that it can be used by the $Migrate$ operator to host VMs, especially those that require large resources.

3.4.2. Exploration with tabu search

TS employs the $Cons\_Ex(i, j; m, n)$ and $Migrate(i, m, n)$ operators to explore the search space. To prevent TS from short-term cycling, a tabu list l is used to avoid revisiting candidate solutions recently encountered. Specifically, after performing the $Cons\_Ex(i, j; m, n)$ move, the involved virtual machine pair of $V_i \in P_m$ and $V_j \in P_n$ is recorded in the tabu list and will not be allowed to be swapped again in the next tt iterations, where tt is a parameter called the tabu tenure.

The pseudo-code of TS is given in Algorithm 3. TS first initializes each element of the tabu list l to be 0 (line 3). Then TS conducts a series of iterations following the best improvement principle (lines 4-24). During each iteration, TS replaces the current solution S with the best admissible neighboring solution S' (ties are broken randomly), in terms of function r from the $Cons\_Ex(i, j; m, n)$ neighborhood (lines 5-6). Then the involved VM pair V_i and V_j is appended to the tabu list (line 7). A move is treated

as admissible if it is not marked as tabu. After that, TS explores the $Migrate(i, m, n)$ neighborhood and if the neighborhood is not empty, TS replaces the current solution S with the best neighboring solution S' having the largest d value (lines 8-10). Otherwise the algorithm examines VMs on the overloaded PMs one by one and attempts to reallocate such a VM to a non-overloaded PM by visiting the PMs in alphabetical order with respect to the capacity of PMs. As soon as a feasible migration that satisfies the capacity constraints of a non-overloaded PM is found, the migration is performed immediately (lines 11-19). If there is no successful migration move performed, the counter h is increased by 1 (lines 20-21). TS terminates when any of the following two conditions is met: 1) a k-machine solution S is found, in this case, S is returned and a success is reported; 2) the search depth u of TS is reached, i.e., the number of iterations for which the migration operation was not executed, in this situation, the final encountered solution S is returned.

3.5. Perturbation procedure

Perturbation is one of the key factors in ILS. After a round of local search, a new solution is generated by changing the current local optimum, which is then used as the starting solution for the next round of local search.

When designing a perturbation strategy, it is crucial to consider the specific characteristics of the problem at hand and ensure that it aligns effectively with the local search algorithm. If the perturbation is too strong, it may behave like a pure random restart. On the other hand, if the perturbation is too weak, the local search may return to the local optimum just visited, and the diversification of the search will be very limited [25].

We design a specific and moderate perturbation strategy for VMP problem. When TS stops, the search is regarded as being trapped in a local optimum and the perturbation procedure is executed. The perturbation procedure first identifies the set of virtual machines V^p located on the overloaded physical machines. Then each virtual machine $V_j \in V^p$ is reallocated to a randomly selected physical machine $P_j \in P^k$. In Section 5.3, we study the usefulness of this perturbation strategy.

4. Experimental results and comparisons

In this section, we report computational results of the proposed algorithm on well-known benchmark instances and provide comparisons with several

state-of-the-art algorithms from the literature [2, 24, 29].

4.1. Benchmark instances

We assess the performance of the proposed algorithm and the reference algorithms on the following three sets of 1800 benchmark instances (see [29]). These 1800 instances are divided into three types based on the homogeneity of the physical machines and whether there is a bottleneck resource: Set A, Set B, and Set C. The instances of Set A, Set B, and Set C were generated in [29]. The Set A instances were created according to the procedure introduced in [35], while the Set B and Set C instances were produced following the method presented in [24]. For each instance type, there are 600 instances including six subsets with different numbers of virtual machines $N_{VM} \in \{100, 200, 300, 400, 500, 1000\}$ and each subset consists of 100 instances, resulting in a total of 1800 instances grouped into 18 subsets.

4.1.1. Set A instances: homogeneous environment without bottleneck

The physical machines of Set A instances are homogeneous. The CPU and RAM capacities of the PMs are both 500 units. The requirements of the VMs are uniformly randomly generated integers in the range [1, 128] for CPU and [1, 100] for RAM, resulting in a slightly larger average CPU requirement than RAM requirement (but still close to 1: 1). In other words, there is no bottleneck resource for Set A instances. Given an instance, the lower bound lb of the number of physical machines k for any optimal solution is calculated as follows [29]:

$$lb = max\{\frac{\sum_{i=1}^{N_{VM}} R_i^{CPU}}{C_A^{CPU}}, \frac{\sum_{i=1}^{N_{VM}} R_i^{RAM}}{C_A^{RAM}}\} \tag{11}$$

where C_A^{CPU} and C_A^{RAM} are the CPU and RAM capacities of the instances, respectively.

4.1.2. Set B instances: homogeneous environment with bottleneck

The physical machines of Set B instances are homogeneous, equipped with 16 units for CPU and 32 units for RAM. The CPU (RAM) requirements of VMs are uniformly randomly generated integers in the range [1, 4] ([1, 8]). As the probability of generating the 4 units of CPU requirement is 0.25, but the probability of yielding the 8 units of RAM requirement is 0.125, CPU is the bottleneck resource. The lower bound of the Set B instances is calculated by the Equation (11).

4.1.3. Set C instances: heterogeneous environment with bottleneck

The instances of Set C simulate the heterogeneous environment with two types of physical machines named as PM_{C_1} and PM_{C_2}. For each instance, PM_{C_2} accounts for only 10% of the PMs, while the remainder of the PMs are PM_{C_1}. A PM_{C_1} is equipped with 16 units of CPU and 32 units of RAM, whereas a PM_{C_2} features 32 CPU units and 128 RAM units. The VM requirements are from the discrete uniform distributions in the range [1, 8] for CPU and [1, 32] for RAM, indicating that the bottleneck resource of Set C is RAM. The lower bound of Set C instances is calculated by [29]:

$$lb = |PM_{C_2}| + max\{\frac{\sum_{i=1}^{N_{VM}} R_i^{CPU} - |PM_{C_2}|C_{C_2}^{CPU}}{C_{C_1}^{CPU}}, \frac{\sum_{i=1}^{N_{VM}} R_i^{RAM} - |PM_{C_2}|C_{C_2}^{RAM}}{C_{C_1}^{RAM}}\} \tag{12}$$

where $|PM_{C_2}|$ is the number of PMs of PM_{C_2}, and $C_{C_1}^{CPU}$, $C_{C_1}^{RAM}$ ($C_{C_2}^{CPU}$, $C_{C_2}^{RAM}$) are the CPU, RAM capacities of PM_{C_1} (PM_{C_2}) respectively.

4.2. Experimental settings and parameter tuning

The proposed algorithm CCILS was implemented in Java language[1]. All experiments were conducted on a computing platform with an Intel Xeon E5-2695 v4 processor (2.10 GHz) and 2 GB RAM under the Linux operating system. Following the reference algorithms [29], CCILS was executed one time for each instance. The cutoff time for a run was set to 5 seconds.

The proposed CCILS algorithm requires only two parameters: the tabu tenure tt and the search depth of the tabu search u. Table 1 displays the candidate and final values of these parameters. The final parameter values were consistently used for all the subsequent experiments. To analyze the roles of these parameters in the algorithm and to evaluate the sensitivity of each parameter, we conducted a one-at-a-time sensitivity analysis. For the experiment, we randomly selected 10 instances out of each instance subset, totaling 180 instances and ran for 10 times per instance. Each series of

[1]The source code of the proposed CCILS algorithm will be made available upon the publication of the paper at `https://github.com/neteasefans/virtual-machine-placement-problem.git`

testing experiments focused on one parameter at a time by varying its value in a pre-determined range while fixing another parameter to the default value shown in Table 1.

Table 1: Settings of the parameters.

Parameter	Section	Description	Considered values	Final value
u	3.4	Search depth of the tabu search	{5, 20, 30, 50,100}	20
tt	3.4	tabu tenure	{5, 20, 50, 100}	20

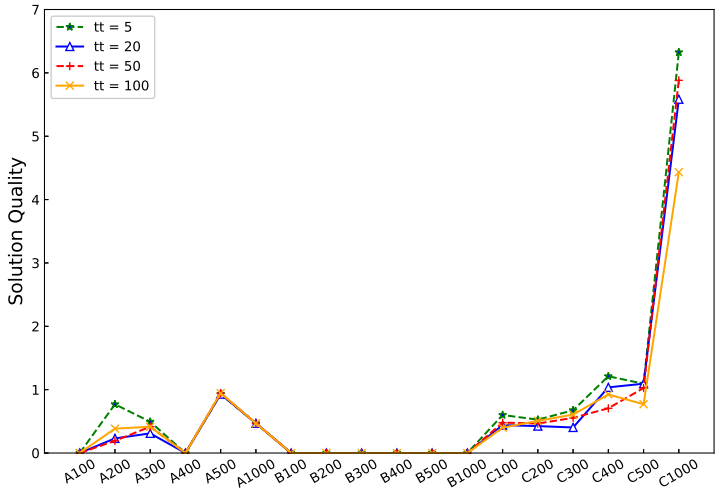

Figure 1: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS with different values of tt.

The results are presented in Fig. 1 and Fig. 2, where the X-axis and Y-axis show respectively the instance subsets and the average solution quality over the randomly selected 10 instances of each subset. For each instance, the solution quality sq is defined as the percentage gap from the best objective value to the lower bound lb calculated by the Equation (11) or (12):

$$sq = 100(\frac{K_S - lb}{lb}) \tag{13}$$

where K_S is the number of used PMs of the obtained best solution S^*. To determine whether different values of a given parameter show statistical

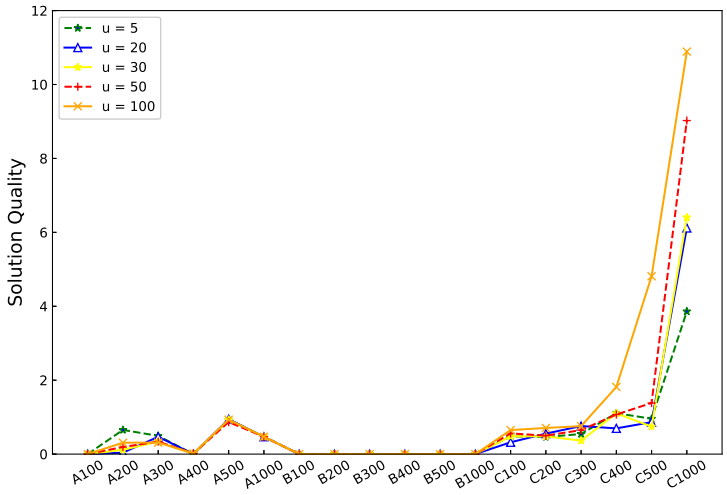

Figure 2: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS with different values of u.

differences in the samples, the non-parametric Friedman tests [13] were conducted. Results of the tests indicate that CCILS is sensitive to the settings of parameters tt (p-value = 1.58e-02) and u (p-value = 4.62e-02). The recommended parameter values from this experiment are 20 for tt and 20 for u.

4.3. Comparison results with state-of-the-art algorithms

Table 2: Comparative results between the proposed algorithm CCILS and the reference algorithms concerning solution quality (%) on the 18 subsets of instances.

	Solution Quality (%)					
Subset	OEMACS[24]	GA1[2]	GA2[2]	PACO(S)[29]	PACO(P)[29]	CCILS
A100	8.98	8.80	8.54	8.25	8.25	**0.00**
A200	4.71	5.13	4.41	4.47	4.47	**0.12**
A300	3.26	4.21	2.92	3.32	3.32	**0.26**
A400	2.78	4.78	2.46	2.78	2.78	**0.00**
A500	2.39	5.54	1.92	2.42	2.42	**0.90**
A1000	1.96	11.30	1.26	1.58	1.58	**0.37**
$\overline{Avg}$	4.01	6.63	3.59	3.80	3.80	**0.28**
B100	8.24	6.49	7.75	6.24	6.24	**0.00**
B200	5.05	3.01	4.78	3.08	3.08	**0.00**
B300	3.72	2.05	5.36	2.11	2.11	**0.00**
B400	3.47	1.64	6.77	1.69	1.69	**0.00**
B500	2.89	1.25	6.30	1.37	1.37	**0.00**
B1000	2.82	0.79	11.40	0.91	0.91	**0.00**
$\overline{Avg}$	4.37	2.54	7.06	2.57	2.57	**0.00**
C100	13.70	6.96	23.90	7.67	7.67	**0.40**
C200	11.90	5.03	31.40	6.10	6.10	**0.46**
C300	11.70	4.46	37.20	5.39	5.39	**0.77**
C400	12.10	4.07	41.30	4.66	4.66	**1.17**
C500	12.00	3.93	43.00	4.44	4.44	**0.93**
C1000	12.80	3.70	54.50	**3.62**	**3.62**	6.78
$\overline{Avg}$	12.37	4.69	38.55	5.31	5.31	**1.75**
$\#Best$	0	0	0	1	1	17
p-value	2.20e-5	1.62e-4	2.20e-5	1.62e-4	1.62e-4	

Table 3: Comparative results between CCILS and the reference algorithms in terms of execution time (in seconds) on the 18 subsets of instances.

	Execution Time (s)					
Subset	OEMACS[24]	GA1[2]	GA2[2]	PACO(S)[29]	PACO(P)[29]	CCILS
A100	0.94	9.10	15.05	0.12	0.05	0.02
A200	12.22	60.23	107.90	0.42	0.15	0.21
A300	51.81	194.40	354.30	0.89	0.27	0.37
A400	158.80	440.00	817.70	1.52	0.50	0.09
A500	369.80	832.80	1575.00	2.39	0.76	2.44
A1000	5451.00	5711.00	12120.00	10.37	2.99	2.41
$\overline{Avg}$	1007.43	1207.92	2498.33	2.62	0.79	0.92
B100	1.08	9.04	15.32	0.12	0.05	0.02
B200	14.14	59.15	109.20	0.42	0.15	0.02
B300	58.75	189.70	358.60	0.90	0.27	0.05
B400	181.10	426.20	829.10	1.51	0.50	0.11
B500	414.80	805.40	1596.00	2.35	0.76	0.14
B1000	6080.00	5547.00	12190.00	9.59	2.90	0.75
$\overline{Avg}$	1124.98	1172.75	2516.37	2.48	0.77	0.18
C100	1.65	9.73	16.98	0.13	0.06	0.07
C200	22.14	63.66	116.70	0.46	0.16	0.39
C300	88.56	205.00	383.70	0.98	0.27	1.20
C400	270.30	456.80	874.70	1.66	0.51	3.90
C500	633.00	858.40	1665.00	2.62	0.77	4.05
C1000	8874.00	5754.00	12490.00	10.69	2.87	5.01
$\overline{Avg}$	1648.27	1224.60	2591.18	2.76	0.77	2.44

To evaluate the performance of the proposed CCILS algorithm, this section provides an extensive computational comparison between CCILS and the reference algorithms including OEMACS [24], GA1 [2], GA2 [2] and PACO [29]. For PACO, contrasting the serial version that uses only one thread, the parallel version calling OpenMP to allocate each construction process (a total of 20) to a separate thread is also developed. To distinguish between parallel and serial versions, they are denoted as PACO(P) and PACO(S) respectively. The numerical results reported by the reference algorithms are directly extracted from [29]. All the reference algorithms were implemented in C++, and compiled by the GNU g++ compiler, with the '-O2' option. The reference algorithms were conducted on a computing platform with an Intel Xeon E5-2640 v4 processor with 20 cores running at a base frequency of 2.40 GHz and a maximum frequency of 3.40 GHz. Note that the stopping conditions of the reference algorithms are not uniform. GA1 and GA2 are terminated after performing a fixed number of 200 generations, while OEMACS and PACO stop after they perform 50 generations.

Due to the differences in computing platforms and termination criteria, giving a completely fair comparison between CCILS and the reference algorithms is a challenging task. Therefore, we mainly focus on the quality of the obtained solutions and provide execution time for indicative purposes

only. We utilize CPU frequency to compare the speed of the processors employed for testing methods. Compared to our processor (Intel Xeon E5-2695 v4, 2.10 GHz), the processor (Intel Xeon E5-2640 v4, 2.40 GHz) used by the reference algorithms has a scaling factor of 1.14, indicating that the processor utilized in this study is slightly slower.

Table 2 reports the computational results of CCILS as well as the reference algorithms in terms of the solution quality on the benchmark instances. Column 'Subset' represents the subset name of instances with different types and sizes. Row '$\overline{Avg}$' shows the average results across the subsets of three instance sets. Row '$\#Best$' represents the number of cases for which an algorithm yields the best result among all the compared methods. To verify whether there are statistical differences in terms of solution quality between CCILS and the compared algorithms, the statistical results from the non-parametric Friedman test are provided in row 'p-value'. The solution quality results reported by each compared algorithm are the average results across all the 100 instances of each subset. The best values are highlighted in bold.

From Table 2, we can observe that the proposed algorithm CCILS consistently demonstrates a superior performance compared to the reference algorithms on the 18 subsets of 1800 instances. Specifically, for the 18 instance subsets, CCILS reports the best results with respect to the average solution quality across 100 instances of each subset on 17 subsets, while OEMACS, GA1, GA2, PACO(S) and PACO(P) obtain the best results for 0, 0, 0, 1, 1 cases, respectively. Notably, the average solution quality value reported by CCILS is 0.00 for the 8 subsets A100, A400, B100, B200, B300, B400, B500, B1000, indicating that the lower bound is achieved for each instance of these subsets, in other words, the optimal solutions are found for all the 800 instances of the 8 subsets. Indeed, we found the optimal solutions for 1390 cases out of all the 1800 benchmark instances[2]. Moreover, the average solution quality values for each of the 18 subsets are quite low, with almost all of them falling below 1%, except for the C400 and C1000 subsets. For the Set C instances with both heterogeneous environment and bottleneck resources, the proposed algorithm exhibits also better performances than all the reference algorithms on C100, C200, C300, C400 and C500 subsets, and

[2]The optimal solution certificates are available at `https://github.com/neteasefans/virtual-machine-placement-problem.git`

worse than GA1 and PACO only on C1000.

Table 3 shows the average execution time in seconds to reach its best result elapsed by each algorithm across all the 100 instances of each subset. Table 3 indicates that in terms of the average execution time, CCILS and PACO (including PACO(S) and PACO(P)) show similar performance while much better performance than OEMACS, GA1 and GA2 for the three sets of instances. For the Set A and Set C instances, CCILS and PACO(S) performs slightly worse than PACO(P), while CCILS shows a slightly better performance than PACO(S) and PACO(P) on Set B instances.

The small p-values (< 0.05) from the non-parametric Friedman test confirm the statistically significant difference between the results of CCILS and those of the compared methods, including OEMACS, GA1, GA2, PACO(S) and PACO(P). Overall, the proposed algorithm is very competitive comparing with the existing VMP algorithms both in solution quality and computational efficiency.

In summary, when comparing with the state-of-the-art VMP approaches, the proposed CCILS algorithm exhibits very competitive performance in both solution quality and computational efficiency for various test graphs. However, the performance of the algorithm is, to some extent, dependent on the dedicated evaluation functions for neighboring solution selection, and designing appropriate evaluation functions for given problems is a challenging task.

4.4. Application to the temporal extension of VMP

Table 4: Comparative results between CCILS and Cplex solver on the VMP-T instance subsets.

	CCILS		Cplex		
Subset	*sol*	*time*(*s*)	*sol*	*time*(*s*)	Gap (%)
Rand_VMPT_10	5.90	0.005	5.60	1.585	5.36
Rand_VMPT_20	12.20	0.004	11.70	66.123	4.27
Rand_VMPT_30	17.20	0.007	15.70	131.761	9.55
Rand_VMPT_40	20.90	0.009	19.80	726.348	5.56
Rand_VMPT_50	27.40	0.007	25.20	724.128	8.73
Rand_VMPT_60	32.50	0.013	29.60	1178.432	9.80
Rand_VMPT_70	35.80	0.017	32.70	1293.783	9.48
Rand_VMPT_80	39.20	0.020	36.10	2201.686	8.59
Rand_VMPT_90	46.60	0.018	42.30	2352.999	10.17
Rand_VMPT_100	52.50	0.013	48.50	1907.893	8.25
Avg	29.02	0.012	26.72	1058.474	7.98

To further expand the application scope of the proposed CCILS algorithm, we extend CCILS to address a temporal extension of VMP [5] called VMP-T, which plays a critical role in practical applications. VMP-T extends VMP by requiring that each VM $i \in \{1, ..., N_{VM}\}$ resides in the system for the duration within the requested time interval $[s_i, e_i]$, where $s_i \in \mathbb{Z}^+ \cup \{0\}$ and $e_i \in \mathbb{Z}^+$ are the start and the end times of VM i respectively, and $e_i > s_i \geq 0$ holds. We define the index set τ to represent the start and end times of virtual machines. Specifically, $l \in \tau$ indicates a time point corresponding to either the start or the end of a VM, forming an ordered set of times, such that $t_l > t_{l-1}$ for $l > 1$. To designate the presence of a VM i at time t_l, we introduce a binary parameter a_{il} that is set to 1 if and only if VM i exists at time $t_l \in \{0, ..., T\}, l \in \tau$ where T denotes the planning horizon. It is assumed that each VM arrives and departs at the beginning of a time period. Therefore, we have $a_{il} = 0$ for $l = e_i$. The mathematical formulation of VMP-T just replaces the constraints (3) and (4) of the model (1)-(6) by the following constraints (14) and (15) to ensure that the total load on a physical machine does not exceed its capacity.

$$\sum_{i=1}^{N_{VM}} a_{il} x_{im} R_i^{CPU} \leq C_m^{CPU} y_m, m \in \{1, ..., N_{PM}\}, l \in \tau_A \tag{14}$$

$$\sum_{i=1}^{N_{VM}} a_{il} x_{im} R_i^{RAM} \leq C_m^{RAM} y_m, m \in \{1, ..., N_{PM}\}, l \in \tau_A \tag{15}$$

where $\tau_A \subseteq \tau$ denotes the index set of VM start times.

To handle the VMP-T problem, we slightly adapted CCILS by only adding the procedure that calculates the total CPU and RAM requirements of the VMs for each start time $l \in \tau_A$.

To evaluate the performance of the proposed CCILS on the VMP-T problem, we made a comparative experiment between CCILS and the IBM ILOG Cplex 12.8.0 solver on 100 randomly generated instances following the work [10]. The 100 instances are grouped to 10 subsets with the number of VMs $N_{VM} \in \{10, 20, ..., 90, 100\}$ and each subset consists of 10 graphs. Specifically, the CPU and RAM capacities of the PMs are both set to 30 units. The requests of the VMs are uniformly randomly distributed between the integer intervals [1,30] for both CPU and RAM. The time duration of the VMs are uniformly randomly generated integers between 10 and 100. Both CCILS and the Cplex solver were executed once for each instance under the

same computing platform as described in Section 4.2, and the time limit for one run is set to 10 seconds for CCILS and 3600 seconds for Cplex.

The computational results of CCILS and the Cplex solver on the ten subsets of VMP-T benchmarks are shown in Table 4. Columns '*sol*', and '$time(s)$' give the average objective value and average runtime in seconds to reach the final solution across all the instance of each subset, respectively. Column '$Gap(\%)$' gives the percentage gap between the objective value obtained by CCILS and Cplex. We can observe that CCILS is able to obtain an acceptable solution within a very short computing time for the instances of 10 subsets, being beneficial for practical applications. If a specialized evaluation function for VMP-T is designed, the efficiency of CCILS to address VMP-T could be further improved. Moreover, from the detailed results shown in Table 9 and 10 of the Appendix, one can see that although the performance of CCILS on VMP-T is a little worse than the Cplex solver on most test graphs, CCILS can still obtain the optimal solutions on 18 instances in a short time.

5. Analysis

In this section, we perform an analysis of three key ingredients of the proposed algorithm to get useful insights of their impacts on CCILS's performance, including the constrained exchange operator, the dedicated evaluation functions and the perturbation procedure. The experiments were conducted on the 180 instances used in Section 4.2, with 10 randomly selected instances from each of the 18 instance subsets.

5.1. Effect of the constrained exchange operator

We observe that the exchange operators adopted by other local searches for VMP mainly aim to exchange VMs between an overloaded PM and a non-overloaded PM. In this work, a novel constrained exchange operator (denoted as $Cons\_Ex(i, j; m, n)$) swapping VMs only between non-overloaded PMs is specifically designed. To examine the efficiency of the $Cons\_Ex(i, j; m, n)$ operator used by the proposed algorithm, we made a comparison between CCILS and its two variants CCILS_{RE} and CCILS_{OE} which use the exchange operators implemented by PACO [29] and OEMACS [24], respectively. The other ingredients are kept unchanged for CCILS_{RE} and CCILS_{OE}. Note that the perturbation procedure is disabled for the underlying local search, i.e., ILS of CCILS, CCILS_{RE} and CCILS_{OE}. For the experiments, the three

methods were independently run 10 times for each instance with a cutoff time of 5 seconds per execution.

Table 5: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS and its two variants using different exchange operators.

	Solution Quality (%)		
Subset	CCILS	CCILS_{RE}	CCILS_{OE}
A100	**0.00**	2.85	2.77
A200	**1.54**	1.92	1.92
A300	**0.47**	0.52	0.52
A400	**0.19**	0.19	0.19
A500	**0.93**	0.95	0.95
A1000	**0.48**	0.54	0.54
$\overline{Avg}$	**0.60**	1.16	1.15
B100	**0.00**	1.25	1.25
B200	**0.03**	2.45	2.54
B300	**0.79**	3.13	3.18
B400	**0.48**	3.22	3.22
B500	**0.29**	2.95	2.95
B1000	**0.05**	3.03	3.03
$\overline{Avg}$	**0.27**	2.67	2.70
C100	**8.50**	23.23	23.32
C200	**9.31**	24.23	24.21
C300	**9.93**	23.47	23.63
C400	**14.23**	25.28	25.25
C500	**13.57**	25.00	25.03
C1000	**13.47**	20.95	20.92
$\overline{Avg}$	**11.50**	23.69	23.73
$\#Best$	18	0	0
p-value		3.70e-5	3.70e-5

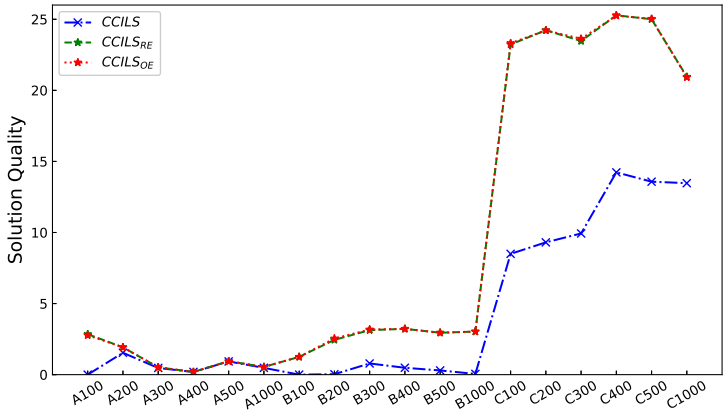

Figure 3: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS and its two variants using different exchange operators.

Table 5 reports the average solution quality obtained by CCILS as well as CCILS_{RE} and CCILS_{OE} on the selected instances. The symbols have the same meanings as before. Table 5 shows that CCILS is clearly superior to CCILS_{RE} and CCILS_{OE} by yielding the best results on all the 18 subsets concerning the average solution quality, while CCILS_{RE} and CCILS_{OE} produce the best result on 0, 0 cases. One can observe that for the Set A instances having homogeneous physical machines without bottleneck, CCILS shows a slightly better performance than CCILS_{RE} and CCILS_{OE} by improving the average solution quality by approximately 0.5%. As the difficulty of the problem increases, such as with Set B instances which are homogeneous but have bottleneck resources, the improvement ratio is increased to be approximately 2.4%. For the most challenging Set C instances in a heterogeneous environment with bottleneck resources, the improvement ratio is even more pronounced, approaching 12%. The advantages of CCILS are more clearly illustrated in Fig. 3. The X-axis and Y-axis of Fig. 3 have the same meanings as in Fig. 1. The p-value from the non-parametric Friedman test in terms of the average solution quality between CCILS and its two variants, further demonstrates the effectiveness of the proposed $Cons\_Ex(i, j; m, n)$ operator in enhancing the overall performance of CCILS.

To further evaluate the impact of the constrained exchange operator exchanging VMs between non-overloaded PMs on computing time, we made a time-to-target experiment between CCILS and its variant CCILS_{OL}. The latter exchanges VMs between an overloaded PM and a non-overloaded PM, while all other components remain unchanged. Both CCILS and CCILS_{OL} terminate upon reaching a predefined target solution quality: for Set A and Set B instances, the target is their lower bound while for Set C instances, it is 1.05 times the lower bound. Additionally, if the target is not achieved within 10 seconds, the algorithm stops.

Table 6: Average execution time in seconds on the 10 randomly selected instances for each subset obtained by CCILS and its variant CCILS_{OL}.

	Execution Time (s)		
Subset	CCILS	CCILS_{OL}	ro
A100	0.01	0.01	1.00
A200	0.25	5.00	20.00
A300	1.31	2.01	1.53
A400	0.23	1.01	4.39
A500	6.00	6.00	1.00
A1000	6.02	7.01	1.16
$\overline{Avg}$	2.30	3.51	1.53
B100	0.01	0.01	1.00
B200	0.02	0.02	1.00
B300	0.08	4.15	51.88
B400	0.10	5.02	50.20
B500	0.12	6.07	50.58
B1000	0.59	9.19	15.58
$\overline{Avg}$	0.15	4.08	27.20
C100	0.03	2.02	67.33
C200	0.06	3.41	56.83
C300	0.15	3.16	21.07
C400	0.53	7.70	14.53
C500	0.82	8.01	9.77
C1000	4.49	~~10.00~~	2.23
$\overline{Avg}$	1.01	5.72	5.66

Table 6 reports the average execution time in seconds to reach the target solution quality for 10 randomly selected instances from each subset. Column 'ro' represents the ratio of the computing times required by CCILS and CCILS_{OL}. If the target is not reached within 10 seconds, the reported time is marked with a deletion line. The experimental results demonstrate that CCILS consistently achieves the target quality with less or equal computation time across 15 benchmark subsets. Notably, its performance advantage is more pronounced in Set B and Set C than in Set A. This observation suggests that exchanging VMs between non-overloaded PMs is particularly effective for addressing VMP problems involving bottleneck resources and heterogeneous infrastructures.

5.2. Impact of the dedicated evaluation functions

To examine the efficiency of the dedicated evaluation functions of the proposed algorithm, we compared CCILS with three variants CCILS_{RS}, CCILS_{GS} and CCILS_{RM}. Given the current solution S, CCILS_{RS} randomly selects a neighboring solution from $N_1(S)$ rather than using the r function, while CCILS_{GS} uses the evaluation function $r^{'}$ to evaluate a candidate solution S' from $N_1(S)$.

$$r^{'}(S^{'}) = L_m^{CPU} + L_m^{RAM} \tag{16}$$

Both CCILS_{RS} and CCILS_{GS} use the same $Migrate$ operator as CCILS. As for CCILS_{RM}, it randomly selects a neighboring solution from $N_2(S)$ rather than using the d function. CCILS_{RM} uses the same $Cons\_Ex(i, j; m, n)$ operator as CCILS. For the experiments, CCILS and the variants were independently run 10 times for each instance with a cutoff time of 5 seconds per execution.

Table 7: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS and its three variants with different evaluation functions.

	Solution Quality (%)			
Subset	CCILS	CCILS_{RS}	CCILS_{GS}	CCILS_{RM}
A100	**0.00**	**0.00**	**0.00**	**0.00**
A200	0.10	1.81	1.15	**0.04**
A300	**0.44**	0.52	0.52	0.49
A400	**0.02**	0.19	0.15	**0.02**
A500	0.95	0.95	0.95	**0.93**
A1000	**0.47**	0.54	0.48	**0.47**
$\overline{Avg}$	**0.33**	0.67	0.54	**0.33**
B100	**0.00**	**0.00**	**0.00**	**0.00**
B200	**0.00**	**0.00**	**0.00**	**0.00**
B300	**0.00**	**0.00**	0.02	**0.00**
B400	**0.00**	**0.00**	0.49	**0.00**
B500	**0.00**	**0.00**	0.41	**0.00**
B1000	**0.00**	**0.00**	1.52	**0.00**
$\overline{Avg}$	**0.00**	**0.00**	0.41	**0.00**
C100	0.08	3.01	**0.04**	0.20
C200	0.51	3.01	**0.41**	0.73
C300	**0.34**	3.51	0.56	0.52
C400	**0.70**	4.47	1.89	1.76
C500	**1.20**	4.58	1.87	1.65
C1000	8.58	**4.60**	11.34	9.11
$\overline{Avg}$	**1.90**	3.86	2.69	2.33
$\#Best$	13	8	5	11
p-value		0.01	0.01	0.10

Table 7 and Fig. 4 present the computational results of CCILS and its variants. One can observe from Table 7 that CCILS, CCILS_{RS}, CCILS_{GS} and CCILS_{RM} report the best results in terms of the average solution quality on 13, 8, 5, and 11 subsets, respectively. By comparing the average solution quality reported by CCILS, CCILS_{RS} and CCILS_{GS} across each instance set given in row '$\overline{Avg}$', CCILS shows a better performance than CCILS_{RS} and CCILS_{GS} (0.33% vs. 0.67% vs. 0.54% for Set A, 0.00% vs. 0.00% vs. 0.41% for Set B, 1.90% vs. 3.86% vs. 2.69% for Set C), reflecting the important role of evaluation function r in CCILS. Although CCILS_{RM} using a different evaluation function to evaluate candidate solutions from the $Migrate$ neighborhood which displays a similar performance with CCILS for the Set A, Set B instances, CCILS consistently reports better solution quality

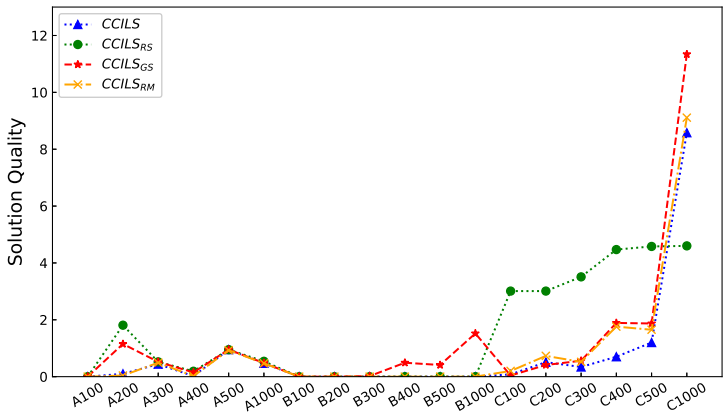

Figure 4: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS and its three variants with different evaluation functions.

than CCILS_{RM} in all the 6 subsets of Set C, disclosing the effectiveness of function d to solve difficult VMP instances. The p-values (< 0.05) from the non-parametric Friedman test in terms of the solution quality reveal a statistically significant difference between CCILS and its two variants CCILS_{RS} and CCILS_{GS}.

5.3. Usefulness of the perturbation procedure

To evaluate the effectiveness of the perturbation procedure of the proposed algorithm, an experiment was conduced to compare the proposed CCILS against two variants: CCILS_{B} using a different perturbation strategy and CCILS_{NP} disabling the perturbation process while keeping other ingredients unchanged. During the perturbation process, CCILS_{B} reallocates each virtual machine to a randomly selected physical machine while CCILS reallocates the virtual machines on overloaded physical machines only. For the experiments, CCILS and its two variants were run 10 times for each instance with a cutoff time of 5 seconds.

Fig. 5 provides the comparative results of CCILS, CCILS_{B} and CCILS_{NP}. It can be seen that CCILS obtains better or same solutions for all the instance subsets. The effect of the perturbation procedure is particularly significant in Set C instances, which have both bottleneck resources and heterogeneous physical machines. This indicates that the perturbation

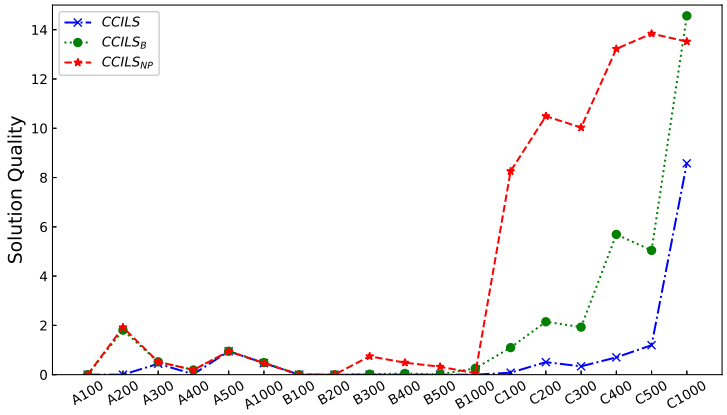

Figure 5: Average solution quality on the randomly selected 10 instances for each subset obtained by CCILS and its two variants.

procedure is suitable for solving VMP problems in complex situations. The superiority of the perturbation procedure was further confirmed by the non-parametric Friedman test with p-values < 0.05.

5.4. Average infeasibility induced by the solution initialization procedure

Table 8: Average infeasibility, expressed as a percentage, on 10 randomly selected instances from each subset.

Subset	Average infeasibility (%)
A100	7.69
A200	3.85
A300	2.60
A400	1.89
A500	3.15
A1000	2.12
B100	18.75
B200	16.24
B300	10.01
B400	5.10
B500	6.55
B1000	3.73
C100	5.03
C200	2.18
C300	1.76
C400	1.37
C500	1.03
C1000	0.50
$\overline{Avg}$	5.20

Recall that given a $(k+1)$-machine solution, the solution initialization procedure (SIP) of Section 3.3 generates an input solution with k physical machines by reallocating randomly each virtual machine located on the physical machine P_{k+1} to a physical machine $P_r\,(r \in \{1, 2, ..., k\})$. This reallocation procedure may lead to an infeasible solution violating CPU or RAM capacity constraint of a physical machine. To assess the infeasibility of a solution induced by SIP, we define the infeasibility as the ratio of the number of physical machines violating either the CPU or RAM capacity to the total number of physical machines k. We performed an experiment to record the average infeasibility of an instance observed during the iterative decomposition of VMP into a sequence of subproblems, in which k is decreased by 1 each time a feasible solution is found. The computational results are presented in Table 8. As shown in Table 8, each subset from the three benchmark sets exhibits a relative low average infeasibility.

6. Conclusions

The virtual machine placement (VMP) problem is an NP-hard problem of great theoretical and practical importance. This work proposes a cardinality constrained iterated local search (CCILS) for VMP by transforming VMP into a series of problems, each focusing on a fixed number k of physical machines. CCILS features a tabu search procedure to efficiently explore the search space by using two new neighborhoods based on the dedicated evaluation functions for neighborhood solution selection, and a specific perturbation procedure to prevent the algorithm from stagnation.

The computational experiments show that the proposed algorithm consistently performs better than the state-of-the-art algorithms on the 18 subsets of 1800 instances. In particular, CCILS produces the best results (upper bounds) on 17 subsets in terms of the average solution quality, while OEMACS, GA1, GA2, PACO produce the best results only on 0, 0, 0, 1 subsets. More importantly, by matching the lower bounds for 1390 out of 1800 instances whose optimality was previously unknown, CCILS proves optimal solutions for these 1390 instances for the first time. Additional experiments investigate the effect of the constrained exchange operator, the special evaluation functions, and the perturbation procedure.

In addition to the reported computational results being useful for future studies of VMP, the publicly available source code of CCILS can be useful to researchers and practitioners for solving practical problems that can be

formulated as a VMP model. We further incorporate the time dimension into the VMP problem to enhance the practical applicability of the proposed method. The design ideas adopted in the proposed algorithm are versatile and can be applied to solve other related problems. To further improve the effectiveness of the perturbation strategy, designing a learning-based perturbation mechanism can be considered. Besides, other aspects of the VMP problem, such as energy consumption and load balancing as well as time dimension are also worth studying.

Acknowledgment

We are grateful to the reviewers for their comments that helped us to improve the paper.

References

[1] Abbasi-khazaei, T., & Rezvani, M. H. (2022). Energy-aware and carbon-efficient VM placement optimization in cloud datacenters using evolutionary computing methods. Soft Computing, 26(18), 9287-9322.

[2] Abohamama, A. S., & Hamouda, E. (2020). A hybrid energy-aware virtual machine placement algorithm for cloud environments. Expert Systems with Applications, 150, 113306.

[3] Alves, M. M., Teylo, L., Frota, Y., & Drummond, L. M. (2018, October). An interference-aware virtual machine placement strategy for high performance computing applications in clouds. In 2018 Symposium on High Performance Computing Systems (WSCAD) (pp. 94-100). IEEE.

[4] Avci, M. (2023). An effective iterated local search algorithm for the distributed no-wait flowshop scheduling problem. Engineering Applications of Artificial Intelligence, 120, 105921.

[5] Aydßn, N., Muter, İ., & Birbil, . İ. (2020). Multi-objective temporal bin packing problem: An application in cloud computing. Computers & Operations Research, 121, 104959.

[6] Belgacem, A., Mahmoudi, S., & Ferrag, M. A. (2023). A machine learning model for improving virtual machine migration in cloud computing. The Journal of Supercomputing, 79(9), 9486-9508.

[7] Brandão, J. (2020). A memory-based iterated local search algorithm for the multi-depot open vehicle routing problem. European Journal of Operational Research, 284(2), 559-571.

[8] Breitgand, D., Epstein, A., Glikson, A., Israel, A., & Raz, D. (2013, October). Network aware virtual machine and image placement in a cloud. In Proceedings of the 9th International Conference on Network and Service Management (pp. 9-17). IEEE.

[9] Caviglione, L., Gaggero, M., Paolucci, M., & Ronco, R. (2021). Deep reinforcement learning for multi-objective placement of virtual machines in cloud datacenters. Soft Computing, 25(19), 12569-12588.

[10] De Cauwer, M., Mehta, D., & O'Sullivan, B. (2016, November). The temporal bin packing problem: an application to workload management in data centres. In 2016 IEEE 28th International Conference on Tools with Artificial Intelligence (pp. 157-164). IEEE.

[11] Dell'Amico, M., Furini, F., & Iori, M. (2020). A branch-and-price algorithm for the temporal bin packing problem. Computers & Operations Research, 114, 104825.

[12] Delorme, M., Iori, M., & Martello, S. (2016). Bin packing and cutting stock problems: Mathematical models and exact algorithms. European Journal of Operational Research, 255(1), 1-20.

[13] Derrac, J., García, S., Molina, D., & Herrera, F. (2011). A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. Swarm and Evolutionary Computation, 1(1), 3-18.

[14] Dillon, T., Wu, C., & Chang, E. (2010, April). Cloud computing: issues and challenges. In 2010 24th IEEE International Conference on Advanced Information Networking and Applications (pp. 27-33). IEEE.

[15] Di, S., Kondo, D., & Cappello, F. (2013, October). Characterizing cloud applications on a Google data center. In 2013 42nd International Conference on Parallel Processing (pp. 468-473). IEEE.

[16] Fernandez-Viagas, V., de Athayde Prata, B., & Framinan, J. M. (2022). A critical-path based iterated local search for the green permutation flowshop problem. Computers & Industrial Engineering, 169, 108276.

[17] Friden, C., Hertz, A., & de Werra, D. (1989). Stabulus: a technique for finding stable sets in large graphs with tabu search. Computing, 42(1), 35-44.

[18] Fukunaga, T., Hirahara, S., & Yoshikawa, H. (2017). Virtual machine placement for minimizing connection cost in data center networks. Discrete Optimization, 26, 183-198.

[19] Galinier, P., Hamiez, J. P., Hao, J. K., & Porumbel, D. (2013). Recent advances in graph vertex coloring. In: Zelinka, I., Snášel, V., Abraham, A. (eds), Handbook of Optimization, vol 38 (pp. 505-528). Springer.

[20] Glover, F. (1989). Tabu search-part I. ORSA Journal on Computing, 1(3), 190-206.

[21] Greenberg, A., Hamilton, J., Maltz, D. A., & Patel, P. (2008). The cost of a cloud: research problems in data center networks. ACM SIGCOMM Computer Communication Review, 39(1), 68-73.

[22] Hbaieb, A., Khemakhem, M., & Jemaa, M. B. (2017, October). Using decomposition and local search to solve large-scale virtual machine placement problems with disk anti-colocation constraints. In 2017 IEEE/ACS 14th International Conference on Computer Systems and Applications (pp. 688-695). IEEE.

[23] Kushchazli, A., Safargalieva, A., Kochetkova, I., & Gorshenin, A. (2024). Queuing model with customer class movement across server groups for analyzing virtual machine migration in cloud computing. Mathematics, 12(3), 468.

[24] Liu, X. F., Zhan, Z. H., Deng, J. D., Li, Y., Gu, T., & Zhang, J. (2016). An energy efficient ant colony system for virtual machine placement in cloud computing. IEEE Transactions on Evolutionary Computation, 22(1), 113-128.

[25] Lourenço, H. R., Martin, O. C., Stützle, T. (2019). Iterated Local Search: Framework and Applications. In: Gendreau, M., Potvin, JY.

(eds), Handbook of Metaheuristics. International Series in Operations Research & Management Science, vol 272 (pp. 129-168). Springer.

[26] Masanet, E., Shehabi, A., Lei, N., Smith, S., & Koomey, J. (2020). Recalibrating global data center energy-use estimates. Science, 367(6481), 984-986.

[27] Máximo, V. R., Cordeau, J. F., & Nascimento, M. C. (2022). An adaptive iterated local search heuristic for the Heterogeneous Fleet Vehicle Routing Problem. Computers & Operations Research, 148, 105954.

[28] Medeiros, J. M. F., Subramanian, A., & Queiroga, E. (2025). Population-based iterated local search for batch scheduling on parallel machines with incompatible job families, release dates, and tardiness penalties. Optimization Letters, 19(1), 193-210.

[29] Peake, J., Amos, M., Costen, N., Masala, G., & Lloyd, H. (2022). PACO-VMP: parallel ant colony optimization for virtual machine placement. Future Generation Computer Systems, 129, 174-186.

[30] Pirabán-Ramírez, A., Guerrero-Rueda, W. J., & Labadie, N. (2022). The multi-trip vehicle routing problem with increasing profits for the blood transportation: An iterated local search metaheuristic. Computers & Industrial Engineering, 170, 108294.

[31] Praveen, S. P., Ghasempoor, H., Shahabi, N., & Izanloo, F. (2023). A hybrid gravitational emulation local search-based algorithm for task scheduling in cloud computing. Mathematical Problems in Engineering, 2023(1), 6516482.

[32] Stillwell, M., Schanzenbach, D., Vivien, F., & Casanova, H. (2010). Resource allocation algorithms for virtualized service hosting platforms. Journal of Parallel and Distributed Computing, 70(9), 962-974.

[33] Tawfeek, M. A., El-Sisi, A. B., Keshk, A. E., & Torkey, F. A. (2014). Virtual machine placement based on ant colony optimization for minimizing resource wastage. In Second International Conference on Advanced Machine Learning Technologies and Applications, Cairo, Egypt, November 28-30, 2014. (pp. 153-164). Springer.

[34] Vu, H. T., & Hwang, S. (2014). A traffic and power-aware algorithm for virtual machine placement in cloud data center. International Journal of Grid & Distributed Computing, 7(1), 350-355.

[35] Wilcox, D., McNabb, A., & Seppi, K. (2011, June). Solving virtual machine packing with a reordering grouping genetic algorithm. In 2011 IEEE Congress of Evolutionary Computation (pp. 362-369). IEEE.

[36] Wu, Q., & Hao, J. K. (2013). An adaptive multistart tabu search approach to solve the maximum clique problem. Journal of Combinatorial Optimization, 26, 86-108.

[37] Xu, J., & Fortes, J. A. (2010, December). Multi-objective virtual machine placement in virtualized data center environments. In 2010 IEEE/ACM International Conference on Green Computing and Communications & International Conference on Cyber, Physical and Social Computing (pp. 179-188). IEEE.

[38] Zhou, Q., Benlic, U., & Wu, Q. (2020). An opposition-based memetic algorithm for the maximum quasi-clique problem. European Journal of Operational Research, 286(1), 63-83.

Appendix

This appendix provides the detailed comparison results between CCILS and the Cplex solver on the virtual machine placement problem with time dimension (VMP-T) benchmark instances which are generated following the work [10]. Both CCILS and Cplex are executed once for each instance on the same computing platform as described in Section 4.2. The cutoff time for a run for CCILS and Cplex is 10 seconds and 3600 seconds, respectively. Column 'Ins.' denotes the instance name. Columns '*sol*', and '$time(s)$' represent the objective value and runtime in seconds to reach the final solution for each instance, respectively. Column '$Gap(\%)$' gives the percentage gap between the objective value returned by CCILS and Cplex. For each instance, if the time limit is reached and the optimal solution is not found by the Cplex, we mark the solution with a symbol "TL". Optimal solutions are marked by the symbol "*".

Algorithm 3: Tabu search algorithm

Input: S: an input solution, tt: tabu tenure, u: the search depth of the tabu search

Output: The repaired solution S

```
1  c ← 0 /* Iteration counter */
2  h ← 0
3  l[i][j] ← 0, i, j ∈ {1, ..., N_VM} /*Initialize each element of the tabu list to 0*/
4  while h ≤ u do
5      Select the best admissible neighboring solution S' from the Cons_Ex(i, j; m, n) neighborhood having the largest r value
6      S ← S'
7      l[i][j] = l[j][i] = c + tt /* Update the tabu list l */
8      if Migrate(i, m, n) neighborhood is not empty then
9          Select the best neighboring solution S' from Migrate(i, m, n) neighborhood with the largest d value
10         S ← S'
11     else
12         for i = {1, ..., N_VM} do
13             if i is loaded on an overloaded PM then
14                 for m = {1, ..., N_PM} do
15                     if m is non-overloaded and i can be reallocated to m without violating the capacity then
16                         Migrate i to m
17                         break
18             if A migration move is successfully performed then
19                 break
20         if There is no successful migration move performed then
21             h ← h + 1
22     if S is a k-machine solution then
23         return S and report success
24     c ← c + 1
25 return S
```

Table 9: Detailed results of VMP-T instances from the CCILS algorithm and Cplex solver (part 1).

	CCILS		Cplex		
Ins.	*sol*	*time*(*s*)	*sol*	*time*(*s*)	Gap (%)
Random_10_1	6	0.004	5*	1.121	20.00
Random_10_2	6	0.008	6*	1.718	**0.00**
Random_10_3	7	0.004	7*	1.307	**0.00**
Random_10_4	6	0.011	6*	1.341	**0.00**
Random_10_5	5	0.010	5*	1.700	**0.00**
Random_10_6	6	0.003	5*	1.718	20.00
Random_10_7	6	0.004	5*	1.382	20.00
Random_10_8	6	0.003	6*	1.406	**0.00**
Random_10_9	5	0.005	5*	1.420	**0.00**
Random_10_10	6	0.003	6*	2.739	**0.00**
Random_20_1	14	0.004	13*	1.958	7.69
Random_20_2	14	0.004	14*	0.961	**0.00**
Random_20_3	13	0.004	13*	329.284	**0.00**
Random_20_4	10	0.004	10*	323.323	**0.00**
Random_20_5	13	0.004	12*	0.887	8.33
Random_20_6	11	0.004	9*	0.945	22.22
Random_20_7	12	0.005	11*	0.841	9.09
Random_20_8	13	0.004	13*	1.206	**0.00**
Random_20_9	11	0.008	11*	1.088	**0.00**
Random_20_10	11	0.004	11*	0.734	**0.00**
Random_30_1	18	0.015	16*	117.703	12.50
Random_30_2	17	0.006	15*	587.152	13.33
Random_30_3	18	0.006	16*	108.493	12.50
Random_30_4	15	0.006	14*	3.143	7.14
Random_30_5	18	0.005	15*	2.243	20.00
Random_30_6	20	0.005	20*	5.455	**0.00**
Random_30_7	17	0.010	15*	2.732	13.33
Random_30_8	14	0.010	13*	376.162	7.69
Random_30_9	17	0.006	17*	1.754	**0.00**
Random_30_10	18	0.005	16*	112.771	12.50
Random_40_1	19	0.015	18*	888.425	5.56
Random_40_2	21	0.011	20*	1508.933	5.00
Random_40_3	19	0.007	17*	527.715	11.76
Random_40_4	25	0.010	25*	174.904	**0.00**
Random_40_5	20	0.006	20*	212.165	**0.00**
Random_40_6	22	0.006	20*	119.402	10.00
Random_40_7	19	0.006	19*	814.063	**0.00**
Random_40_8	20	0.019	18*	2269.995	11.11
Random_40_9	24	0.007	23*	641.707	4.35
Random_40_10	20	0.007	18*	106.168	11.11
Random_50_1	27	0.006	25*	1383.902	8.00
Random_50_2	27	0.007	23*	1050.548	17.39
Random_50_3	25	0.006	22*	3.079	13.64
Random_50_4	27	0.008	26*	508.381	3.85
Random_50_5	31	0.008	28*	1647.418	10.71
Random_50_6	26	0.007	24*	213.451	8.33
Random_50_7	27	0.007	24*	217.668	8.33
Random_50_8	33	0.007	32*	420.652	3.13
Random_50_9	25	0.007	24*	1163.428	4.17
Random_50_10	26	0.007	24*	632.756	8.33

Table 10: Detailed results of VMP-T instances from the CCILS algorithm and Cplex solver (part 2).

	CCILS		Cplex		
Ins.	*sol*	*time*(*s*)	*sol*	*time*(*s*)	Gap (%)
Random_60_1	33	0.010	31*	981.145	6.45
Random_60_2	33	0.010	30*	354.121	10.00
Random_60_3	30	0.009	27*	933.424	11.11
Random_60_4	34	0.009	31*	603.154	9.68
Random_60_5	32	0.014	30*	552.331	6.67
Random_60_6	35	0.040	32*	1004.103	9.38
Random_60_7	30	0.009	27*	214.168	11.11
Random_60_8	30	0.008	26*	2960.645	15.38
Random_60_9	37	0.009	35*	1060.863	5.71
Random_60_10	31	0.013	27*	3120.364	11.11
Random_70_1	36	0.011	33*	458.996	9.09
Random_70_2	35	0.054	32*	743.992	9.38
Random_70_3	30	0.019	26*	1061.875	15.38
Random_70_4	37	0.018	35*	413.088	5.71
Random_70_5	37	0.013	34*	404.819	8.82
Random_70_6	36	0.016	33*	1598.851	9.09
Random_70_7	38	0.012	34*	2082.889	11.76
Random_70_8	41	0.012	39*	1347.559	5.13
Random_70_9	34	0.012	31(TL)	3623.407	9.68
Random_70_10	34	0.011	30*	1202.349	16.67
Random_80_1	40	0.030	39*	1360.297	2.56
Random_80_2	39	0.033	37*	1258.880	5.41
Random_80_3	41	0.019	38*	438.903	7.89
Random_80_4	34	0.012	31*	1063.888	9.68
Random_80_5	38	0.012	32(TL)	3635.459	18.75
Random_80_6	39	0.037	37(TL)	3664.330	5.41
Random_80_7	41	0.014	37*	1903.617	10.81
Random_80_8	40	0.020	37*	1418.908	8.11
Random_80_9	40	0.012	38(TL)	3669.510	5.26
Random_80_10	40	0.017	35(TL)	3603.070	14.29
Random_90_1	42	0.015	37*	484.009	13.51
Random_90_2	54	0.022	51*	1821.162	5.88
Random_90_3	40	0.019	37(TL)	3724.944	8.11
Random_90_4	46	0.018	39(TL)	3603.999	12.82
Random_90_5	47	0.022	43(TL)	3604.437	9.30
Random_90_6	49	0.021	43(TL)	3602.191	13.95
Random_90_7	50	0.014	48*	3449.187	4.17
Random_90_8	52	0.016	49*	909.105	6.12
Random_90_9	44	0.021	39*	573.412	12.82
Random_90_10	42	0.013	37*	1757.541	13.51
Random_100_1	54	0.014	50*	175.929	8.00
Random_100_2	58	0.014	55(TL)	3697.406	5.45
Random_100_3	46	0.014	42(TL)	3601.969	9.52
Random_100_4	55	0.014	49(TL)	3602.632	12.24
Random_100_5	47	0.014	43*	343.548	9.30
Random_100_6	55	0.013	52*	1153.631	5.77
Random_100_7	55	0.013	50*	236.343	10.00
Random_100_8	57	0.015	53(TL)	3601.910	7.55
Random_100_9	46	0.012	43*	1217.633	6.98
Random_100_10	52	0.014	48*	1447.932	6.25