Network-aware virtual machine migration in an overcommitted cloud

Article *in* Future Generation Computer Systems · March 2016

DOI: 10.1016/j.future.2016.03.009

CITATIONS
34

READS
275

4 authors, including:

Weizhe Zhang
Harbin Institute of Technology
128 PUBLICATIONS **715** CITATIONS

SEE PROFILE

Huixiang Chen
University of Florida
13 PUBLICATIONS **138** CITATIONS

SEE PROFILE

Future Generation Computer Systems 76 (2017) 428–442

Contents lists available at ScienceDirect

Future Generation Computer Systems

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

Network-aware virtual machine migration in an overcommitted cloud

Weizhe Zhang *, Shuo Han, Hui He, Huixiang Chen

School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China

HIGHLIGHTS

- Network-aware virtual machine migration is formulated into an NP-complete problem.
- A genetic algorithm is proposed to solve this problem.
- An artificial bee colony algorithm is firstly proposed to solve this problem.

ARTICLE INFO

Article history:
Received 10 August 2015
Received in revised form
8 February 2016
Accepted 14 March 2016
Available online 24 March 2016

Keywords:
Cloud computing
Network-aware virtual machine migration
Network communication costs
Migration costs
Genetic algorithm
Artificial bee colony algorithm

ABSTRACT

Virtualization, which acts as the underlying technology for cloud computing, enables large amounts of third-party applications to be packed into virtual machines (VMs). VM migration enables servers to be reconsolidated or reshuffled to reduce the operational costs of data centers. The network traffic costs for VM migration currently attract limited attention.

However, traffic and bandwidth demands among VMs in a data center account for considerable total traffic. VM migration also causes additional data transfer overhead, which would also increase the network cost of the data center.

This study considers a network-aware VM migration (NetVMM) problem in an overcommitted cloud and formulates it into a non-deterministic polynomial time-complete problem. This study aims to minimize network traffic costs by considering the inherent dependencies among VMs that comprise a multi-tier application and the underlying topology of physical machines and to ensure a good trade-off between network communication and VM migration costs.

The mechanism that the swarm intelligence algorithm aims to find is an approximate optimal solution through repeated iterations to make it a good solution for the VM migration problem. In this study, genetic algorithm (GA) and artificial bee colony (ABC) are adopted and changed to suit the VM migration problem to minimize the network cost. Experimental results show that GA has low network costs when VM instances are small. However, when the problem size increases, ABC is advantageous to GA. The running time of ABC is also nearly half than that of GA. To the best of our knowledge, we are the first to use ABC to solve the NetVMM problem.

1. Introduction

Cloud computing is a newly emerged pay-as-you-go utility model that is currently receiving considerable attention from academia and industry. Virtualization, which is the abstraction of logical resources from their underlying physical resources, serves as the technology support for cloud computing. Big data centers comprise large amounts of third-party applications. By packing these applications into virtual machines (VMs), virtualization enables these third-party applications to be reconsolidated onto other servers according to server-side resources and workloads, thereby contributing to significant cost savings.

The most commonly used server consolidation method is energy-aware VM migration, which aims to consolidate as many VMs into servers as possible to improve the utilization of each server and to switch the unused servers to a low-power state. As a result, energy can be saved. This method considers only server-side constraints, including central processing unit (CPU), memory, and storage capacity. The network costs incurred by VM migration are often overlooked.

Network communication consumes a large portion of the total operational costs of data centers. Heller et al. [1] analyzed that a network consumed 10%–20% of the total energy of a data center.

* Correspondence to: Box 320, School of Computer Science and Technology, Harbin Institute of Technology, Harbin, 150001, China.
E-mail address: wzzhang@hit.edu.cn (W. Zhang).

http://dx.doi.org/10.1016/j.future.2016.03.009
0167-739X/

The overall power consumed by networking components in 2006 in the United States itself was 3 billion kWh. The report suggested that data centers suffered from long-lived congestion caused by core network overcommitment and unbalanced workload placement. A good VM migration algorithm can greatly improve network performance and scalability. However, only a few studies presently focus on the network-aware VM migration (NetVMM) problem. Most of them have focused on only one factor of a network, i.e., communication costs, migration costs, or energy consumption.

As for the NetVMM problem, three aspects of problems need to be considered. First, most current approaches for NetVMM consider only server-side resource constraints (CPU, memory, and storage capacity). The communication patterns among VMs are often overlooked. Recent research has shown that the traffic among VMs in a typical Internet data center accounts for approximately 80% of its total traffic. Second, VM migration only considers dynamic consolidation, for which one or some of the hosts of VMs can be changed in response to the change in workloads. However, VM migration also consists of static consolidation, which may last longer. Webmasters prefer the static approach of setting VMs into the optimal state. Third, apart from communication costs, VM migration costs, which refer to the data traffic of one VM migrating from one host to another, are often overlooked.

In this study, we conduct a comprehensive analysis of the NetVMM problem, which is a non-deterministic polynomial time (NP)-complete problem. We consider the measured communication dependencies among VMs in the application tier, the underlying network topology of a data center, and server-side resource constraints. We view the NetVMM problem as a global optimization goal. The optimization goal is to ensure a good trade-off between minimizing the communication costs and the VM migration costs in the data center. We also adopt genetic algorithm (GA) and artificial bee colony (ABC) algorithm to solve the problem. After comparing four types of swarm intelligence algorithms, namely, GA, ant colony optimization (ACO), particle swarm optimization (PSO), and ABC, we conclude that GA has better network costs when VM instances are small. Nevertheless, when the problem size increases, ABC has more advantages than GA, as well as more advantages in running time.

The remainder of this paper is organized as follows: Section 2 introduces the related work. Section 3 presents the formal problem definition of NetVMM. Section 4 elucidates the static VM migration and its swarm intelligence solutions. Section 5 is the experimental section. Section 6 concludes this paper.

2. Related work

VM migration has become a hot research topic in recent years. This topic can be extracted into a scheduling problem. We review the related work from the following aspects:

2.1. From the perspective of energy consumption

Extensive research has focused on saving the energy of data centers by considering system resources, mainly in CPU utilization, and other resources, including memory, disk storage, etc. Beloglazov et al. [2] proposed four energy-aware allocation heuristics that provided the CPU resources of a data center to client applications to improve the energy efficiency of the data center without violating the negotiated service-level agreements (SLAs). In their algorithm, VMs were sorted in decreasing order of their current CPU utilization. Each VM was assigned to a host that offered a minimal increase in power consumption for this allocation. In the other three algorithms, they proposed a method of setting the upper and lower thresholds and kept the CPU utilization of all hosts between these thresholds. If the CPU utilization was below the set thresholds, then all VMs on the host were migrated to other hosts, and this host was switched to fall asleep. If the CPU utilization was above the set thresholds, then some VMs were migrated until the CPU utilization was below the upper threshold. This method decreased the SLA violations by replying consolidations when the utilizations of VMs increased suddenly. Berral et al. [3] proposed a machine learning approach, in which they used the models learned from previous behavior to predict power consumption levels, CPU loads, and SLA timing. This machine learning method handled situations in which information could be missing or unclear. However, CPU utilization is not a good indicator of power consumption if applications on servers change. Dynamic server consolidation should also consider other system metrics, such as memory, input/output, network activity, and disks [4]. Chou et al. [5] proposed the first scheduling algorithm in the context of energy saving for storage systems. They formulated energy-aware scheduling as an NP-complete optimization problem and proposed both optimal and heuristic algorithms for offline, batch, and online cases. They analyzed the complexity of offline scheduling and batch scheduling problems, and proposed that they could be solved by a weighted independent set and a weighted set-cover algorithm. In the online scheduling problem, a cost function was proposed to consider the trade-off between energy and performance. Srikantaiah et al. [6] studied the consolidation for CPU and disk utilization. They evaluated the effect of consolidation with these two resources by varying both CPU and disk utilizations. They found that the energy consumption per transaction resulted in a "U"-shaped curve and that an optimal combination of the two resources existed.

Verma et al. [7] studied the problem of power-aware application placement on heterogeneous virtualized server clusters. They formulated their application controller pMapper to minimize the power consumption and migration costs while meeting the SLA. On the basis of the observation of the use of the first-fit decreasing (FFD) heuristic in the bin packing problem, they proposed three algorithms from three aspects. First, the min power parity was designed to minimize power. Second, the incremental FFD algorithm was a power-minimizing placement algorithm that aimed to migrate as few VMs as possible. Third, the pMaP algorithm took a balanced view of both power and migration costs and aimed to minimize total costs. However, their work did not handle strict SLA requirements [6].

Goudarzi et al. [7] studied the problem of minimizing the power and migration costs in a data center by considering the SLA required by clients. Their main idea was to consider the trade-off between energy costs and client satisfaction specified by SLA. They set the goal of their optimization by including the total energy costs and the penalty of violating SLA and solved their problem by using a heuristic algorithm based on convex optimization and dynamic programming.

All the aforementioned works considered only the energy consumption of system resources (including CPU, memory, disk, etc.). They did not consider the effect of network resources.

2.2. From the perspective of VM migration costs

Verma et al. [7] studied the costs of VM migration. From their experiments, they found that the costs were independent of the background load and depended only on VM characteristics. The costs could be estimated by quantifying the decrease in throughput because of live migration and by estimating the revenue loss caused by the decreased performance. Lefèvre et al. [8] stated that VM live migration consisted of transferring its memory image from the host physical node to a new node. If the VM was running, then it was stopped for a period of time during the copy of the last

memory pages. Liu et al. [9] considered live migration overhead for the performance and energy management of virtualized data centers. By conducting experiments on the XEN platform in a wired network environment, they experimentally verified that migration energy consumption was proportional to the data volume of network traffic caused by VM migration.

2.3. From the perspective of NetVMM

Research on the NetVMM problem is only in its initial stages. We review previous works from the following aspects:

(1) Saving network energy. Heller et al. [1] stated that a network consumed 10%–20% of the total power of a data center. They proposed a new topology underlying data center, i.e., ElasticTree, and formulated a subset and flow assignment problem to minimize network energy. Mann et al. [10] presented VMFlow, which was a framework for the placement and migration of VMs that considered both network topology and network traffic demands to reduce network power while satisfying as many network traffic demands as possible. They transformed VM migration into a flow assignment problem and proposed a fast heuristic algorithm. Experiments showed that VMFlow could reduce 15%–20% of network power and meet 5–6 demands in contrast to recently proposed methods. Zhani et al. [11] considered the problem of allocating both servers and data center networks to multiple virtual data centers (VDCs) and proposed a VDC planner to maximize the revenue of VM migration and to minimize the total energy costs.

(2) Avoiding network congestion in a data center. Wen et al. [12] studied the problem of VM migration to reduce network congestion and transformed it into a quadratic bottleneck assignment problem. They proposed an online consolidation algorithm, VirtualKnotter, with controllable VM migration traffic and low time complexity. Experiments showed that VirtualKnotter had only 5%–10% of network traffic compared with existing methods. Kliazovich et al. [13] presented a scheduling method called data center energy-efficient network-aware scheduling (DENS), which combined energy efficiency and network awareness. DENS attempted to avoid hotspots within a data center while minimizing the number of computing servers required for job execution.

(3) NetVMM that considered the interdependencies of VMs. A recent study [14] has shown that the traffic among VMs in a typical Internet data center accounted for approximately 80% of its total traffic. Thus, the interdependencies of VMs should be considered in NetVMM. Meng et al. [15] studied the effect of network resources on optimizing VM migration on host machines. During their measurement study in operational data centers, they found that the distribution of VMs in data centers was extremely uneven; VM pairs with a relatively heavy traffic rate tended to constantly exhibit a high rate, whereas low pairs tended to exhibit a low rate. They formulated the traffic-aware VM mapping problem as an NP-hard optimization problem and proposed a two-tier approximation algorithm to solve this problem. Experimental results showed that their algorithm could increase the performance and greatly decreased the communication costs of a network. Jayasinghe et al. [16] aimed to enhance the performance and availability of the services located on infrastructure-as-a-service clouds and studied the structural constraint-aware VM placement. The VM migration algorithm aimed to minimize communication costs while meeting the availability and communication constraints. They formulated the problem into three NP-complete problems, namely, (1) VM clustering to optimize communication costs, (2) VM clusters to server rack mapping, and (3) VM-to-physical machine (PM) mapping. However, all the aforementioned works did not consider the server-side constraints, including CPU, memory, and storage capacity.

In response to the shortcomings of previous works, Shrivastava et al. [17] studied the overloaded VM migration problem in consideration of network traffic and server-side constraints. They formulated a data center into a model that considers the inter-VM dependencies and the underlying network topology into VM migration decisions. They proposed a greedy algorithm called AppAware, which assigned VMs to servers one at a time and aimed to minimize the costs resulted from that step. The AppAware algorithm was considered a dynamic VM migration algorithm because it aimed to consider the optimal migration destination of each VM one by one. Thus, we could change one or some of the VMs in response to the changes in workload. However, AppAware neglected the costs of VM migration. In fact, the migration energy consumption is proportional to the data volume of network traffic caused by VM migration [1].

Zhang et al. [18] also studied the overloaded VM migration and proposed a new VM migration algorithm called Scattered, which also considered the interdependencies of VMs and the underlying topology of PMs. This algorithm aimed to minimize communication costs and VM migrations. Nevertheless, X. Zhang et al. did not analyze the data traffic of VM migration.

Chen et al. [19] proposed a new algorithm called migration with link and node load consideration (MWLAN) to improve network traffic load and resource utilization while meeting server-side resource constraints. In MWLAN, each VM migrated according to the revenue of each VM migration and the data traffic of VM migrating from one to another as heuristic information. Experiments showed that MWLAN could decrease the congestions in a data center and increase data transfer rate.

The traditional energy-aware server consolidation method that aimed to improve the utilization of each server would cause oversubscription of resources. Ghorbani et al. [20] studied how to keep the network bandwidth of a data center and transformed this problem into a sequence planning problem. Results showed that their heuristic algorithm could greatly decrease SLAs.

On the basis of all the recent works concerning NetVMM, no one has considered the interdependencies of VMs and the underlying topology of a data center, and has aimed to ensure a trade-off between the communication and migration costs of a data center.

2.4. Swarm intelligence algorithms for VM migration

VM migration and server consolidation problems can be transformed into an NP-complete optimization problem. Finding an optimal solution in polynomial time is not feasible. Swarm intelligence algorithms aim to find an approximation solution through multiple iterations, which are good methods for solving NP-complete problems. Thus, they can be widely used in solving the VM migration and server consolidation problems.

Swarm intelligence algorithms include such algorithms as simulated annealing, tabu search, iterated local search, GA, ACO, PSO, and ABC optimization. The use of swarm intelligence algorithms to solve many combinatorial optimization problems has been a subject of research for a long time. They are widely used to solve NP-complete problems, including traveling salesman problems, task assignment problems (TAPs), scheduling problems, and subset problems. Braun et al. [21] studied the problem of optimally mapping independent real-time tasks onto a set of heterogeneous machines, which was a type of TAP. Given that finding the best heuristic in a specific problem was difficult, a collection of 11 heuristics were implemented and analyzed for their specific TAP. GA consistently achieved the best performance in time and

makespan. In accordance with this previous study, Chen et al. [22] proposed a newly extended ACO and concluded from experiments that ACO outperformed GA in time performance. They also compared their performances in terms of energy consumption and found that the extended ACO achieved an average of 15.8% of energy saving. Agrawal et al. [23] solved the problem using GA by transforming the problem into a vector packing problem and achieved a trade-off between minimizing the number of servers and maximizing the packing efficiency of the utilized server. We also leveraged the automatic memory control and resource scheduling of multiple VMs for cloud computing [24,25].

3. Problem definition

A set of VMs in a data center are represented by $V = \{V_1, V_2, V_3 \ldots V_n\}$ [17], among which O VMs are overloaded, i.e., $O = \{V_1, V_2 \ldots V_o\}$. A set of servers are defined as $S = \{S_1, S_2, S_3 \ldots S_m\}$.

The relation of VMs is represented by a dependency graph $G = (V, E)$, where V is a set of VMs, and E is a set of edges defined as $E = \{(V_i, V_j) |$ where V_i and V_j has communication$\}$. The weight of two vertexes in G is defined as $W(V_i, V_j)$, which is directly proportional to the traffic transferred between V_i and V_j.

The resource requirement (CPU, memory, and storage requirement) of V_i is defined as $Load(V_i)$, and the available server-side resource capacity of server S_j is $Capacity(S_j)$. We consider the costs of migrating a pair of VMs that have communication dependency, i.e., the cost of migrating V_i and V_j to servers S_k and S_l is $Cost(V_i, S_k, V_j, S_l) = Distance(S_k, S_l) \times W(V_i, V_j)$. $Distance(S_k, S_l)$ is defined as the latency, delay, or number of hops between servers S_k and S_l. We define a variable X_{ik}. When V_i is placed onto S_k, X_{ik} is 1; otherwise, X_{ik} is 0.

$$X_{ik} = \begin{cases} 1 & \text{if } V_i \text{ is assigned to } S_k \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

$X_{ik}^{jl} = X_{ik} * X_{jl} \cdot X_{ik}^{jl}$ is 1 if V_i and V_j are placed on servers S_k and S_l, respectively; otherwise, X_{ik}^{jl} is 0.

The total communication cost in the data center is defined as

$$Cost\_Comm = \sum_{A} Cost(V_i, S_k, V_j, S_l) \times X_{ik}^{jl}, \tag{2}$$

where A is defined as $A = \{(i, j, k, l) | i < j, k < l, j < |V|, l < |S|\}$. From Eq. (2), if the dependent VMs are placed close to one another in the topology of the server, then the total network traffic will be greatly reduced. The problem constraints are as follows:

$$\sum_{k}^{m} X_{ik} = 1, \quad \forall\, i,\ V_i \in O. \tag{3}$$

Eq. (3) indicates that each overloaded VM V_i must be placed on exactly one server S_k, and each given VM can only reside on one server at a time.

To meet the server-side resource constraints, the following equation must hold:

$$\sum_{i}^{n} Load_i \times X_{ik} \le Capacity_k, \quad \forall\, k,\ S_k \in S. \tag{4}$$

Eq. (4) ensures that the accumulated load on each server should be less or equal to the capacity of that server, which is the server-side resource constraint.

The assignment of non-overloaded V_i to server S_k is defined as x_{ik}. The following constraints indicate that the assignments of non-overloaded VMs to servers are fixed.

$$x_{ik} = C. \tag{5}$$

Apart from the communication costs in the data center, the costs of VM migration are a significant factor to be considered. Liu et al. [9] experimentally verified that the migration energy consumption was proportional to the data volume of network traffic caused by VM migration. In this study, we interpret the migration costs incurred by data transferring as the multiplier of the VM size and the distance between the source PM and the destination PM. VM migration includes the VM memory and its database for database-related applications, thereby avoiding residual loads in the source site.

To be specific, we assume that the size of V_i is $Size_i$, and it migrates from S_l to S_k. The migration cost is then shown as follows:

$$Cost\_Mig\ (V_i, S_k) = Size_i \times Distance\ (S_l, S_k)\ . \tag{6}$$

The migration costs incurred by data transfer are closely related to the VM size and its associative database. VM is a software package in essence and puts all virtual hardware resources, operating systems, and corresponding programs or databases into a package. When a VM migrates from one host to another, the migration incurs the costs of transferring related data and memory mirroring. Memory mirroring is the VM size. When a VM migrates from the source server to the destination server, the distance between the source and the destination is also a factor to be considered. A far distance denotes considerable hops that are needed to migrate the VM from the source to the destination, thereby leading to a high migration cost.

The cost of migrating V_i from server S_l is represented by

$$Cost\_Mig(V_i) = \sum_{k \in S} Size_i \times D_{lk} \times X_{ik}. \tag{7}$$

S_l is the source server. We consider the situation that V_i migrates to any server S_k. If $X_{ik} = 1$, then V_i is placed onto Sk. $D_{lk} = Distance(S_l, S_k)$.

The total migration cost in the data center is

$$Cost\_Mig = \sum_{i \in O} \sum_{k \in S} Size_i \times D_{lk} \times X_{ik}. \tag{8}$$

To consider both the communication and migration costs in our optimization framework and to improve the scalability and adaptability of our model, we standardize $Cost\_Comm$ and $Cost\_Mig$ into [0, 1], as follows. The standardization of $Cost\_Comm$ is

$$\begin{aligned} Cost\_Comm\_std &= \frac{\sum_{A} Cost(V_i, S_k, V_j, S_l) \times X_{ik}^{jl}}{e_{Topo\_vm} \times \max_{A}(Cost)} \\ &= \frac{\sum_{A} cost(V_i, S_k, V_j, S_l) \times X_{ik}^{jl}}{e_{Topo\_vm} \times \max_{i,j \in V}(W(V_i, V_j)) \max_{k,l \in S}(D_{kl})}, \end{aligned} \tag{9}$$

where e_{Topo_vm} represents the number of VM ($VMNum$) pairs that have communication demands, i.e., the number of edges in the VM dependency graph. Each edge in the dependency graph has $Cost(V_i, S_k, V_j, S_l)$. $Cost\_Comm$ is divided by the number of edges, the maximum communication demand, and the maximum PM distance to standardize the communication costs.

The standardization of $Cost\_Mig$ is

$$Cost\_Mig\_std = \frac{\sum_{i \in O} \sum_{k \in S} Size_i \times D_{lk} \times X_{ik}}{N_O \times \max_{i \in O}(Size_i) \max_{l,k \in S}(D_{lk})}, \tag{10}$$

where $N_O = |O|$, which is the number of overloaded VMs. $\max(Size_i)$ is the largest size of all VMs.

To adjust the weights of communication and migration costs and to ensure scalability and adaptability, we use two weight

coefficients α and β in the optimization function to minimize the total cost *Cost_Total* as follows:

$$Cost\_Total = \alpha \times Cost\_Comm\_std + \beta \times Cost\_Mig\_std. \quad (11)$$

The NetVMM problem is a variant of a multiple-knapsack problem, which is NP-complete [26]. Thus, finding an optimal solution in polynomial time is not practical. Our goal is to find an approximation solution to this NP-complete problem.

4. Discussion about swarm intelligence algorithms

Our optimization framework aims to find the mapping of VMs to servers that minimizes the global network utility value under Constraints (3)–(5). For each overloaded VM from *O*, AppAware selects a destination server that satisfies Constraints (3)–(5); thus, the migration cost is minimized. However, this approach is only a local optimization; it does not consider network traffic a global goal.

From the perspective of the problem itself, a VM set is placed onto servers as a whole. We regard the value in Eq. (11) as a fitness value and minimize the solution fitness. The VM migration algorithm needs to find a solution as approximate as possible to determine the best allocation scheme of the entire VM set. Thus, swarm intelligence algorithms are good choices. In essence, they aim to find a good solution through multiple generations of randomly generated feasible solutions. In this way, the solution fitness is close to the optimal fitness.

5. Applying GA to VM migration

GA is a search algorithm for solving optimization problems; it is an evolutionary algorithm to mimic the phenomenon of population evolution. This phenomenon includes genetics, natural selection, hybridization, and variation. In GA, a population of strings (chromosomes) that encode candidate solutions to an optimization problem evolves toward better solutions.

5.1. General process of GA

GA is a heuristic algorithm that generates solutions to optimization problems using approaches inspired by natural evolution, including inheritance, selection, crossover, and mutation. The evolution generally starts from a population of randomly generated individuals and is an iterative process, with the population in each iteration called a generation. Each individual is called a chromosome. Each chromosome has a fitness value, which is the objective function in the optimization problem being solved.

A population is made up of many chromosomal sequences. GA simulates the populations that continue to process toward the direction of better optimization solutions. In the evolutionary process, each chromosome can perform crossover and mutation operations.

In each new generation, the new population (i.e., the newly generated individuals) is created using the information from the previous ones relying on three operators, namely, *Selection*, *Crossover*, and *Mutation*. The general process of GA is as follows:

General process of GA

```
1:  Initial population generation: generate the initial solutions of population, and evaluate
    the fitness of each chromosome
2:  While (stopping criteria not met){
3:      Selection;
4:      Crossover;
5:      Mutation;
6:      Evaluation;
7:  }
```

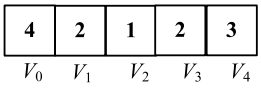

Fig. 1. Example of the chromosome in GA.

1. The initial population starts from a set of randomly generated individuals. Sometimes, algorithm implementers can artificially intervene in the initial population to ensure the quality of initial solutions to obtain a better final solution. Each chromosome in the new population is then evaluated. This process is called *Evaluation*, which involves calculating the fitness of each chromosome.
2. GA enters the main loop, thereby generating a new population from the old one. During the *Selection* process, a part of individuals should be selected from the old population as the new population according to a certain mechanism. A better fitness value of the chromosome means that it has a greater chance of being selected into the new population. The roulette wheel selection method is usually used in the *Selection* process.
3. After selecting two chromosomes in the *Selection* process, *Crossover* and *Mutation* are performed. The two chromosomes generate two new children according to the crossover rate, and the children are mutated based on the mutation rate. The two new individuals are put into the new generation. The process of *Selection–Crossover–Mutation* is repeated until the new generation is the same size as the old one. The Evaluation process is then performed, and the best individual in the new generation is selected as the current best solution.
4. Step 2 is repeated until the stopping criteria are met. The stopping criteria typically consist of the following: (1) the limit of iteration counts is reached; (2) the optimal solution is found; (3) the best fitness value cannot be changed even if further evolution is performed, which is called early convergence; (4) constraints of computing resources (time limit and the memory consumed) exist; (5) human intervention occurs; (6) criteria (1)–(5) are combined.

5.2. Using GA to solve NetVMM

To apply GA to the NetVMM problem, our GA operates on a population of *PopSize* chromosomes. Each chromosome is defined as an $n \times 1$ vector, and the entry in position i is the server onto which V_i is going to map. The mapping for non-overloaded VMs is fixed.

Fig. 1 shows an example of the chromosome for the NetVMM problem. The solution in the figure shows that V_0 is placed onto S_4, V_1 is placed onto S_2, V_2 is placed onto S_1, V_3 is placed onto S_2, and V_4 is placed onto S_3. This structure is the same for the four swam intelligence algorithms, i.e., representing the structure of ant in the ACO, the particle in the PSO, and the bee in the ABC.

The fitness value of chromosome *Solution* equals to the value of Eq. (11), i.e.,

$$f(Solution) = \alpha \times Cost\_Comm\_std + \beta \times Cost\_Mig\_std. \quad (12)$$

5.3. Crossover and mutation

We use the most widely applied *Crossover* and *Mutation* methods by Jong et al. [27], namely, typical two-point *Crossover* and bit-flop *Mutation*. Figs. 2 and 3 show two examples to explain *Crossover* and *Mutation*.

Typical two-point *Crossover*: Crossover enables children to inherit the good parts of parents to generate better solutions than their parents have. For each pair of father and mother $Solution_a$ and $Solution_b$, *Crossover* is performed according to the probability

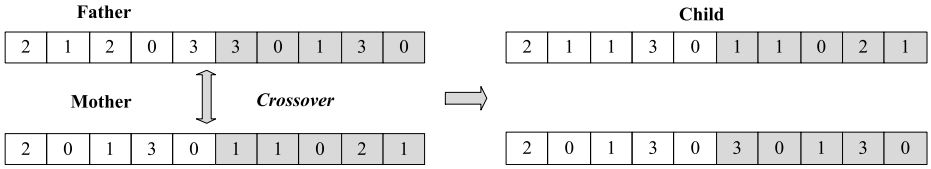

Fig. 2. Example explaining the typical two-point *Crossover*.

crossover rate. A random number k is then generated, i.e., $Solution_a$ and $Solution_b$ exchange their corresponding part from index k, and $Solution_c$ and $Solution_d$ are obtained as the children. Fig. 2 shows an example that explains the typical two-point *Crossover*. A random position number between 1 and 10 is first generated; in this study, the result is 6. The corresponding part of pair parents from the 6th position is then exchanged to generate another pair (see the change in the darks).

Bit-flop *Mutation*: After crossover, each chromosome mutates based on the probability *mutation rate*. A random number k is generated, and the value in the kth position is changed randomly to another server number. Bit-flop *Mutation* is distinct for its no-more-than-one-time mutation for each chromosome. Fig. 3 shows an example that explains the bit-flop *Mutation*. A random position number between 1 and 10 is initially generated; in this study, the result is 3. The corresponding part of the parents of the 3rd position is subsequently randomly changed to generate another (see the change in the darks), i.e., another server is randomly selected for VM_3 allocation, except SM_2; in this study, the result is SM_1.

5.4. GA-based VM migration (GAVMM) algorithm

The GAVMM algorithm is shown as follows:

Algorithm 1: GAVMM

Input: V=$\{V_1,V_2\ldots V_n\}$, S=$\{S_1,S_2\ldots S_m\}$, O=$\{V_1,V_2\ldots V_o\}$, $Load(V_i)$=$Load_i$, $Capacity(S_j)$=$Capacity$
Mappings C: $V_i \rightarrow S_k$, $V_i \in V$, $S_k \in S$, Dependency graph G=(V, E) with weights W
Network distance $Distance(S_k, S_l)$
Output: $Solution_{best}$ //the best allocation of VM to PM, $BestFitness$=f($Solution_{best}$)

Let *PopSize* be the number of chromosomes in the population; let *nIterations* be the total iterations of GA; let *nStagCount* be the count of iterations; when *nStagCount* iterations show no improvement in $Solution_{best}$, GA stops;
$Solution[i]$ (i=1,2…*PopSize*) indicates the i^{th} *chromosome* in the population, $Solution[i][j]$ indicates the destination PM of the jth VM of $Solution[i]$. $fitness[i]$=$f(Solution[i])$

Step 1: initialization of the population
1: $Solution_{App} \leftarrow$ *Application-Aware*( ) // generate the solution of AppAware
2: $Solution[1] \leftarrow Solution_{App}$
3: for each $Solution[i]$, $\forall\, i \in \{2,\ldots popsize\}$
4: Randomly generate $Solution[i]$
5: $fitness[i] \leftarrow f(Solution[i])$
6: if $fitness[i]$<*BestFitness* then
7: $Solution_{best} \leftarrow Solution[i]$
8: *BestFitness*←$fitness[i]$
9: end if
10: end for
Step 2: main loop
11: *iter*←0, *stagCount*←0
12:while *iter*<*nIterations* and *stagCount*<*nStagCount* do
13: *newSolutions*←NULL
14: while *newSolutions*.size( )<*popSize* do
15: $Solution_a \leftarrow$RoulettewheelSelection(*Solution*, *popsize*)
16: $Solution_b \leftarrow$RoulettewheelSelection(*Solution*, *popsize*)
17: ($Solution_c$, $Solution_d$)←Crossover($Solution_a$,$Solution_b$)
//$Solution_c$ and $Solution_d$ are the children of $Solution_a$ and $Solution_b$ by *crossover*
18: $Solution_c \leftarrow$Mutation($Solution_c$)
19: $Solution_d \leftarrow$Mutation($Solution_d$)
20: *newSolutions*←push($Solution_c$)
21: *newSolutions*←push($Solution_d$)//put children into the new population
22: end while
23: *Solution*←*newSolutions* //update the population
24: $Solution_{iterbest} \leftarrow$FindWithMinFitness(*Solution*)
//obtain the chromosome that has the smallest fitness value
25: if $fitness(Solution_{iterbest})$<$fitness(Solution_{best})$ then
26: $Solution_{best} \leftarrow Solution_{iterbest}$
27: else *stagCount*←*stagCount*+1
28: end if
29: *iter*←*iter*+1
30:end while
31:**Step 3**: output $Solution_{best}$ as the best VM allocation

To adopt a guided mechanism to ensure the quality of the initial solution algorithm and to maintain the diversity of the population by randomly initializing other chromosomal sequences, in the population initialization, we select the solution of the AppAware algorithm as the first chromosome of the population, whereas the rest of the genome sequence is randomly generated. After initialization, the fitness value for each chromosome is calculated, and the optimal solution $Solution_{best}$ is selected. In the iterative process, we first adopt the roulette wheel selection method uniformly and randomly select two individuals as parents. We then perform crossover according to a certain *crossover rate* and generate two sub-individuals, namely, $Solution_c$ and $Solution_d$.

The crossover strategy is explained as follows: For $Solution_a$ and $Solution_b$, a chromosomal index k is randomly selected, and the subsequent sequence from index k is exchanged. Mutation is performed in accordance with a certain percentage (we use *bit-flop mutation* as our mutation strategy). The new variation individuals $Solution_c$ and $Solution_d$ are finally obtained as the parents of the next round. With alternating crossover and mutation operations, the number of new populations will finally reach *PopSize*.

The fitness of each chromosome in the new population is calculated. In this step, LocalSearch is abandoned. Our numerous experiments show that for GA, using LocalSearch causes a solution space to become limited, thereby resulting in a relatively poor quality of the optimal solution compared with the one without LocalSearch for GA.

$Solution_{iterbest}$, as the best individual, needs to be selected among a new population, and its fitness is compared with that of $Solution_{best}$ as the global optimum individual. If the former is greater, then $Solution_{best}$ is updated. This loop continues until *nIterations* time cycles are reached, or a better $Solution_{best}$ cannot be obtained within *nStagCount* generations.

5.5. Time complexity analysis

We consider the time complexity of only one generation in GA. The time complexity is $O(Selection) + O(Crossover) + O(Mutation)$. To be specific, $O(Mutation)$ is $O(PopSize * n)$; $O(Crossover)$ is $O(PopSize{*}n)$; $O(Selection)$ is the time complexity of roulette wheel selection, which is $O(PopSize)$. Thus, the total time complexity is $O(PopSize * n)$. n is the *VMNum*.

5.6. Determining the optimal parameters of GA

The parameters of GA are important to its performance. The key parameters are *crossover rate*, *mutation rate*, and *PopSize*.

Crossover rate decides the frequency of crossover. If no crossover occurs, then the children are all copies of parents; otherwise, the children inherit a part of their parents. For example, in case that the *crossover rate* is 100%, all offspring are produced by the crossing over of parents; if the *crossover rate* is 0%, then all offspring are a direct copy of parents. The goal of setting the crossover rate is to enable the children to inherit the good parts of parents to achieve a better fitness value.

Mutation rate decides the frequency of mutation. If no mutation occurs, then chromosomes are generated directly after crossover without any changes; otherwise, parts of chromosomes

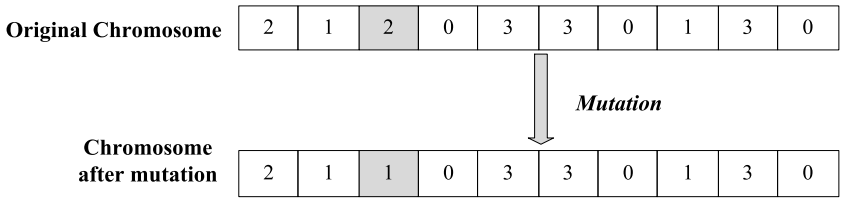

Fig. 3. Example explaining the bit-flop *Mutation*.

Table 1
Most widely used GA parameters.

PopSize	*crossover rate*	*mutation rate*	Crossover method	Mutation method
50	0.6	0.001	Typical two-point *Crossover*	Bit-flop *mutation*

Table 2
Parameter setting for determining *PopSize*.

PopSize	*crossover rate*	*mutation rate*	*nIterations*
20–100, 20 one step	0.6	0.1	5000

can be changed. For example, if the *mutation rate* is 100%, then the entire chromosome will change; if the *mutation rate* is 0%, then no change will occur. Setting the *mutation rate* prevents the early convergence of GA, i.e., GA converges into a search space that is not the optimal solution. However, mutation may not occur often because it will turn GA into random search.

PopSize is defined as the number of chromosomes in a population. If the population size is small, then GA will have a low chance to perform a crossover operation; thus, the solution search space will be partly searched. By contrast, if the population is large with many chromosomes, then GA execution will become slow. Therefore, selecting an appropriate population size is crucial.

Accordingly, Jong et al. [27] provided the most widely used GA parameters, as shown in Table 1.

However, the searching space of our problem is so large that a small population may not give a good approximation solution. Galletly [28] stated that the mutation rate was closely related to the optimization problem itself. The principle of determining the *mutation rate* is to ensure that 30%–40% of chromosomes are unchanged until the algorithm stops. However, a significantly high *mutation rate* will prompt GA to search randomly, thereby preventing the algorithm from converging [27]. We experimentally determine the *mutation rate* and *PopSize* and set other parameters as shown above.

We test the effects of *mutation rate* and *PopSize* on the algorithm performance by changing a parameter once to determine the optimal parameters of GA and to explore how *PopSize* and *mutation rate* influence the algorithm performance.

(1) **Determining *PopSize***

We experimentally determine that *PopSize* varies from 20 to 100. The parameter setting at a step of 20 is shown in Table 2.

The fitness and elapsed time of GA are shown in Table 3.

Table 3 shows that fitness decreases with an increase in *PopSize*. As *PopSize* increases, the elapsed time of GA also increases; hence, we need more individuals of the population to perform GA. We use "*" to represent the best situation under different *VMNum*. When *PopSize* is 100, three situations occur (*VMNum* = 60, 80, 100) in which 100 is the optimum. With the expanding scale, the advantages of *PopSize* of 100 become much obvious. For the other two situations, the fitness of *PopSize* = 100 is also close to the best.

Table 3
Fitness and elapsed time under different PopSize.

PopSize		*VMNum*				
		20	40	60	80	100
20	Fitness	0.251	0.309	0.297	0.282	0.274
	Time	2.569	7.159	10.57	38.96	59.25
40	Fitness	0.238	0.307	0.288	0.277	0.246
	Time	4.654	13.55	24.94	96.10	187.0
60	Fitness	**0.234***	0.299	0.285	0.280	0.236
	Time	7.877	21.40	42.06	120.0	274.8
80	Fitness	0.2473	**0.297***	0.284	0.265	0.234
	Time	9.073	27.33	64.51	186.2	366.1
100	Fitness	0.248	0.309	**0.281***	**0.261***	**0.233***
	Time	11.823	30.43	32.94	218.3	382.2

Table 4
Parameter setting for determining mutation rate.

PopSize	*Crossover rate*	*Mutation rate*	*nIterations*
100	0.6	0–0.1, 0.02 one step	5000

Considering that fitness is more important than elapsed time, we use 100 as the optimal *PopSize*.

(2) **Determining the optimal *mutation rate***

We set *PopSize* = 100, and *mutation rate* varies from 0.0 to 0.1 at a step of 0.02. The parameter setting for determining *mutation rate* is shown in Table 4.

If the *mutation rate* is small, then GA cannot guarantee the diversity and quality of solutions; if the *mutation rate* is large (larger than 0.4 in this study), then GA is equivalent to random search, which cannot maintain the outstanding individuals in the process of changing from one iteration to the next and guarantee the optimal quality and convergence of the algorithm. Therefore, the optimal *mutation rate* needs to be ascertained by experiments.

Table 5 shows that a clear difference exists between GA with mutation and no mutation. When *VMNum* is 20, the fitness is 0.3798 if the *mutation rate* is 0, and the best fitness is 0.301; when *VMNum* is 40, the fitness is 0.333 if the *mutation rate* is 0, and the best fitness is 0.209; when *VMNum* is 60, the fitness is 0.4098 if the *mutation rate* is 0, and the best fitness is 0.328; when *VMNum* is 80, the fitness is 0.3759 if the *mutation rate* is 0, and the best fitness is 0.229; when *VMNum* is 100, the fitness is 0.3779 when the *mutation rate* is 0, and the best fitness is 0.209. We use "*" to represent the best situation under different *VMNum*. A *mutation rate* of 0.1 is the optimum because it contains three best situations.

From the perspective of elapsed time, as *mutation rate* increases, elapsed time first presents an increasing trend; as *mutation rate* reaches 0.1, it reduces significantly. Consequently, we set the optimal *mutation rate* to 0.1.

6. Applying ABC to VM migration

In nature, a scouter performs a waggle dance to notify others about the distance to, the direction to, the quantity of, and the

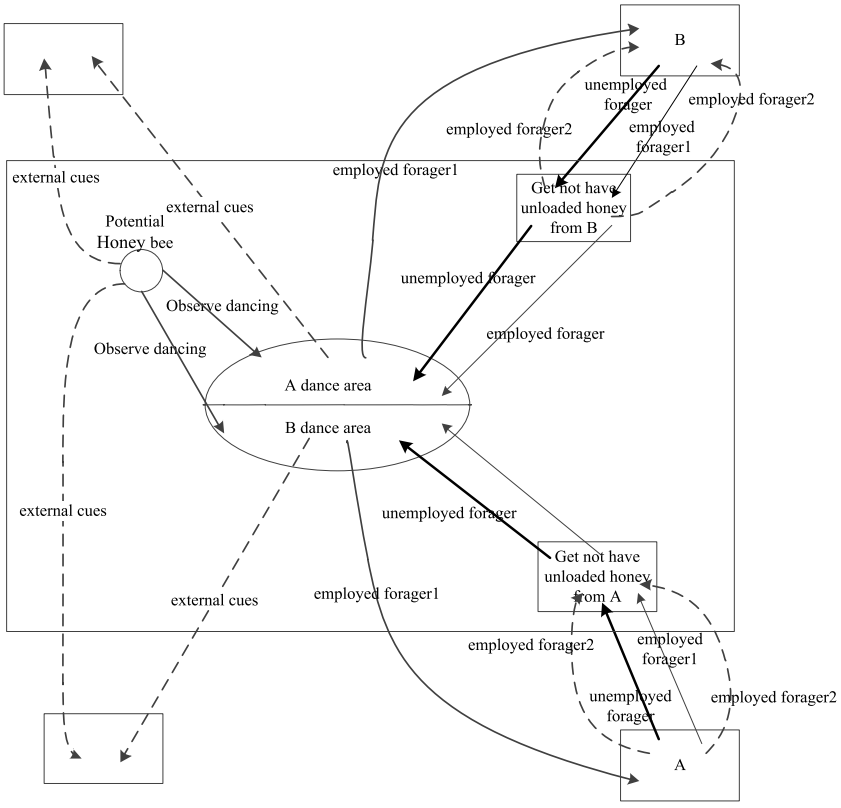

Fig. 4. Honey-collecting behavior.

quality of a food source. For example, if the scouter dances fast, then the food source is relatively short.

Inspired by the food collection behavior and mating behavior of bees, early scholars designed corresponding bee colony algorithms to solve complex computational problems. In general, the colony algorithm is classified into two models, namely, honey and mating. The former is more widely used than the latter. Tereshko et al. developed a model of the foraging behavior of a honeybee colony on the basis of reaction–diffusion equations [29]. This model, which led to the emergence of collective intelligence of honeybee swarms, consisted of three essential components, namely, food sources, employed foragers, and unemployed foragers. Two leading characteristics of the honeybee colony behavior were also defined, namely, recruitment to a food source and abandonment of a source. The main components of this model are explained as follows [29]:

1. Food sources: In an effort to select a food source, a forager bee assesses a number of attributes concerning the food source, including its closeness to the hive, the richness of the energy, the taste of its nectar, and the ease or difficulty of taking out this energy.
2. Employed foragers: An employed forager can be hired at a specific food source that she is presently exploiting. She holds information with regard to this unique source and offers it to other bees hanging around in the hive. The information consists of the distance to, the direction to, and the profitability of the food source.
3. Unemployed foragers: Any forager that looks for a food source to exploit is labeled unemployed. An unemployed forager is normally either a searcher who looks up into environmental surroundings aimlessly or an overlooker who endeavors to locate a food source through the information provided by the employed bee. The mean number of scouters is 5%–10%.

Table 5
Fitness and time under different mutation rates.

Mutation rate		*VMNum*				
		20	40	60	80	100
0	Fitness	0.3798	0.333	0.4098	0.3759	0.3779
	Time	6.38	14.51	30.452	56.194	120.85
0.02	Fitness	0.3151	0.226	0.3487	0.2386	0.2250
	Time	15.948	43.32	101.43	287.73	394.62
0.04	Fitness	0.3104	0.212	0.3356	0.2365	0.2139
	Time	12.474	36.62	94.738	255.04	339.94
0.06	Fitness	0.3102	0.214	0.3341	0.2345	0.2153
	Time	11.314	31.39	82.272	234.24	377.87
0.08	Fitness	0.3094	**0.209***	0.3390	0.2374	**0.209***
	Time	10.614	30.44	80.888	233.32	312.23
0.10	Fitness	**0.301***	0.2258	**0.328***	**0.229***	0.2091
	Time	11.652	38.872	79.844	190.55	291.11

The exchange of knowledge among bees is the most important event to acquire collective information. By observing the information shared by others, each onlooker can select an optimal food source for food collection. An employed forager itself wants an onlooker to share its information by a certain probability, which has a positive correlation with the amount of food sources.

Fig. 4 explains the relationship between bees and food. This figure implies that the dancing area is the most important part of the hive for knowledge exchange. The dancing area controls the quality of food sources by ensuring communication among bees. The related dance is called a waggle dance.

A candidate forager first acts as an unemployed forager. No knowledge exists about the food source around the nest. The bee

performs the following two possible actions:

1. A scouter begins to search around the nest for a food source given some internal knowledge or an external thread.
2. An onlooker begins to search for a food source after watching the waggle dances.

When the food source is found, the bee will mark the location and start exploiting it. The bee then becomes an employed forager. The foraging bee takes a load of the nectar from the source and returns to the hive to unload the nectar to a food store. After unloading, the bee has the following possible actions:

1. It becomes an unemployed forager after abandoning the food source.
2. It dances and recruits nestmates before returning to the food source (employed forager 1).
3. It continues to forage at the food source without recruiting bees (employed forager 2).

All employed foragers start foraging simultaneously. The experimental result shows that the new honeybees are proportional to the difference between the final total numbers of bees and the current honeybees.

6.1. General process of ABC

In ABC, every food source is a possible solution of an optimization problem. The nectar amount of a food source is equal to the quality (fitness) of the solution. The number of employed bees or onlooker bees is the same as the number of solutions (food source) in a population.

ABC first randomly creates an initial population P as SN solutions (food source positions). SN denotes the number of employed bees and onlooker bees, and each solution $Solution_i$ ($i = 1, 2 \ldots SN$) is an n-dimensional vector. n is the number of optimization parameters.

After the initial stage, the algorithm begins to enter the cycle phase. Employed foragers begin to search according to the information they obtain and modify the next food source they may collect, which is called the feasible solution. The fitness of a new feasible solution and the old one is calculated and compared. If the new one is larger than the old one, then the bees remember the new feasible solution and discard the old one. Otherwise, the bees will continue to remember the old feasible solution.

After the search stage of the employed foragers, the bees will share their information on the food source with the onlooker bees. After evaluating the information, the onlookers select an optimal feasible solution. The behavior of the onlookers is similar to that of the employed foragers; they calculate the fitness of the new feasible solution and the old one and select the one with higher fitness as the new feasible solution.

After initialization, the population of solutions is subjected to repeated cycles, i.e., $C = 1, 2 \ldots maxCycle$. Each cycle consists of the search process of **employed foragers**, **onlooker**, and **scouter**. (1) Each employed forager modifies its food source according to the local knowledge and tests the fitness of the new solution. If the new solution has better fitness than the previous one, then the bee replaces the old one using the new; otherwise, the bee retains the old one. (2) After the searching process of all employed bees, they share the fitness information (nectar information) of the food sources with the onlooker bees. Each onlooker selects the food source on the basis of all fitness values after the searching process of the employer bees. Better fitness denotes a higher probability that the food source will be selected. The onlooker then produces a new solution through modifications as in the case of the employed bee. If the new solution has better fitness, then the onlooker bee replaces its old solution using the new one. (3) When an old food source is abandoned after *limit* times of trying, the scouter attempts to discover new food sources randomly.

In the first stage of the cycle, bees enter the hive and share the information they know about a food source with the onlookers waiting for information from the dance area. The bees who select a food source in accordance with what they acquire in the dancing area are defined as the onlooker bees, whereas the bees who share the information are defined as employed foragers. After sharing information, each employed forager continues to search for honey on the basis of the information he obtains with regard to food sources, searches for new food sources, and selects a better food source by comparing the merits among food sources.

In the second stage of the cycle, the onlooker selects a food source on the basis of the information from the dancing area shared by employed foragers. A high abundance of the honey food source corresponds to a high probability that it will be selected. When the onlooker reaches a selected place, it evaluates the food source and selects a better food source.

In the third stage of the cycle, the scouter selects a new food source randomly to replace the abandoned one.

Each cycle generally comprises one onlooker to search food sources randomly, and the quantity of employed bees and onlooker is equal. These three stages will loop until a maximum number of iterations *maxCycle* is reached.

The main process of ABC is as follows:

1: Initialize Population
2: **repeat**
3: Place the employed foragers on their food sources.
4: Place the onlooker bees on the food sources depending on their fitness value.
5: Send the scouter bees to the search area to discover new food sources.
6: **until** stopping requirements are met.

In the searching process of the onlooker [process (2)], each food source in the population is selected with a probability of p_i, i.e.,

$$p_i = \frac{fitness_i}{\sum_{k=1}^{SN} fitness_k}, \tag{13}$$

where $fitness_i$ is the fitness value of $solution_i$. SN is the number of solutions in the population, which also equals the number of employed and onlooker bees.

In the searching process of employed bees [process (1)], each employed bee modifies its solution on the basis of the following equation:

$$FoodNew_{ij} = Food_{ij} + \theta_{ij} \left(FoodNew_{ij} - Food_{kj}\right). \tag{14}$$

$\forall i \in \{1, 2 \ldots SN\}$, and $\forall j \in \{1, 2 \ldots n\}$. $Food_k$ is selected randomly from the population, but it has to be different from $Food_i$. θ_{ij} is a randomly generated real number between $[-1, 1]$. This number controls the production of the neighboring food source around $Food_{ij}$ and visually represents the comparison of two food positions by a bee. Eq. (14) indicates that as the difference between the parameters of $Food_{ij}$ and $Food_{kj}$ decreases, the perturbation on the position $Food_{ij}$ also decreases. Thus, as the search approaches the optimal solution in the search space, the step length is adaptively reduced.

The scouter is in charge of abandoning an old food source with a newly generated one. In ABC, if a solution cannot be improved through processes (1) and (2) in a predetermined number of cycles, then that food source is assumed to be abandoned. The value of the predetermined number of cycles is an important control parameter in ABC, which is called *limit*. Assuming that the abandoned source

is $Food_i$ and $j \in \{1, 2 \ldots n\}$, the scouter discovers a new food source to replace $Food_i$ using the following operator:

$$Food_ij = Food^j_{\min} + rand[0, 1](Food^j_{\max} - Food^j_{\min}). \quad (15)$$

When a new food source $Food_{ij}$ is generated, it will be compared with the old one in terms of the honey degree. If the new one is higher, then it will replace the old.

The detailed process of ABC is as follows:

Detailed process of ABC

begin
initialize $Food_i$, i=1,2,…,*SN*
calculate the fitness of $Food_i$
set *cycle*=1
while (*cycle*<*MCN*){
    In the stage of employed forager, use Equation (14) to produce a solution $Food_i$
    Employed forager selects a better solution by greedy selection.
    Calculate P_i for x_i selected using Equation (13).
    In the stage of onlooker, select x_i based on P_i, and produce a new solution $Food_i$ using Equation (14).
    The onlooker selects a better $Food_i$ by greedy selection.
    The scouter checks whether any solutions exceed *limit*; if so, produce a random solution using Equation (15).
    Remember the optimal solution.
    cycle=*cycle*+1
end while
end

To sum up, ABC uses four selection processes as follows: (1) the onlooker uses a global probability to select a possible food source by using Eq. (13); (2) the employed and onlooker bees search food sources in their head by using Eq. (14) and find a new food source near the current food source where they are; (3) the employed and onlooker bees select a better food source by comparing the old and the new ones through greedy selection and abandon the worse one; (4) the scouter randomly searches for another food source by using Eq. (15).

The most important parameters that control the performance of ABC are the number of food numbers *SN*, *limit*, and the total iterations *maxCycle*.

6.2. Applying ABC for solving NetVMM

(1) Solutions are constructed as follows. For the application of ABC to the NetVMM problem, each food is a solution, which represents the VM placement onto PM. The food structure is the same as the chromosome in GA (Fig. 1). Each food is defined as an $n \times 1$ vector, and the entry in position i is the server onto which V_i is mapped. The mapping for non-overloaded VMs is fixed. During initialization, a food source will produce a random allocation for VM mapping to SM until all VMs are allocated, which we call the feasible solution, or no more server resources can be used, which is called infeasible solution.

(2) The fitness value of Food equals the value of Eq. (11), i.e.,

$$f(Food) = \alpha \times Cost\_Comm\_Std + \beta \times Cost\_Mig\_std. \quad (16)$$

(3) A food source for the employed and onlooker bees is modified as follows: In each iteration, the employed and onlooker bees will both modify the food source and select a better one with higher fitness by using Eq. (14). (See Fig. 5.)

An example of one modifying operation for a food source is shown. *Food* [*i*] is the selected food source, and *Food* [*k*] is a randomly generated food source ($k = 1, 2 \ldots FoodNumber$, $k \neq i$). A random position *j* is generated, and *FoodNew* [*j*] is obtained using Eq. (17). If the fitness of *FoodNew* is less than that of *Food* [*i*] (i.e., $f(FoodNew) < f(Food[i])$), then *FoodNew* is updated for *Food* [*i*].

$$FoodNew[j] = Food[i][j] + \theta ij\,(Food[i][j] - Food[k][j]) \quad (17)$$

$$\theta ij = RandDouble[-1, 1].$$

The detailed algorithm procedure is shown as follows:

Algorithm 2: ABC-based VM Migration

Input: $V=\{V_1,V_2\ldots V_n\}$, $S=\{S_1,S_2\ldots S_m\}$, $O=\{V_1,V_2\ldots V_o\}$, $Load(V_i)=Load_i$, $Capacity(S_j)=Capacity$
Mappings *C*: $V_i \rightarrow S_k$, $V_i \in V$, $S_k \in S$, Dependency graph $G=(V, E)$ with weights *W*
Network distance $Distance(S_k, S_l)$
Output: *GlobalMin* //the best allocation of VM to PM, $BestFitness=f(GlobalMin_t)$

FoodNumber=*SN*, which is the number of foods in the population
trial[*i*](*i*=1,2…*Popsize*), each *Food*[*i*] corresponds to a *trial*[*i*] that represents the times when no better food source is obtained
Step 1: initialization of the population
1: **for each** *Food*[*i*], $\forall i \in \{1,2\ldots FoodNumber\}$
2: Randomly generate *Food*[*i*]
3: *fitness*[*i*]←*f*(*Food*[*i*])//calculate the fitness of *Food*[*i*]
4: **if** *fitness*[*i*]<*BestFitness* **then**
5: *GlobalMin*←*Food*[*i*]
6: *BestFitness*←*fitness*[*i*]
7: **end if**
8: **end for**
Step 2: main loop
9: *iter*←0
10: **while** *iter*<*maxCycle* **do**
//process (1), searching of employed bees:
11: *j*←RandInteger(1, *n*)
12: *k*←RandInteger(1,*FoodNumber*) and $k \neq i$
13: generates food source *FoodNew* based on Equation (14)
14: **if** *fitness*(*FoodNew*)<*fitness*(*Food*[*i*]) **then**
15: *Food*[*i*]←*FoodNew*
16: **else** *trial*[*i*]←*trial*[*i*]+1//if no new food, update *trial*[*i*]
17: **end if**
//process(2), searching of onlooker bees:
18: *foodCount*←1//used for counting the number of modified foods
19: **while** *foodCount*≤*FoodNumber* **do**
20:*Food*[*tmp*]←RoulettewheelSelection(*Food*,*FoodNumber*)
21: Produces modifications on *Food*[*tmp*] as the case of employed bees, in 11–13
22: **if** *fitness*(*tmpFoodNew*)<*fitness*(*Food*[*tmp*]) **then**
23: *Food*[*tmp*]←*tmpFoodNew*
24: **else** *trial*[*tmp*]←*trial*[*tmp*]+1
25: **end if**
26: *foodCount*←*foodCount*+1
27: **end while**
//process (3), searching of scouter bee
28: **for each** *Food*[*i*], $\forall i \in \{1,2\ldots FoodNumber\}$
29: **if** *fitness*[*i*]<*BestFitness* **then**
30: *BestFitness*←*fitness*[*i*]
31: *GlobalMin*←*Food*[*i*]
32: **end if**
33: **if** *trial*[*i*]>*limit* **then**
34: *Food*[*i*]←RandomGenerate()//generate based on (15)
35: **end if**
36: *iter*←*iter*+1
37: **end while**
Step 3: output the best solution
38: output *GlobalMin* as the best VM allocation

We initially generate a random food source community, which is the main idea of ABC, to maintain the diversity of food sources. After initialization, we calculate the fitness value for each food source and select the optimal solution *GlobalMin*. In the iteration

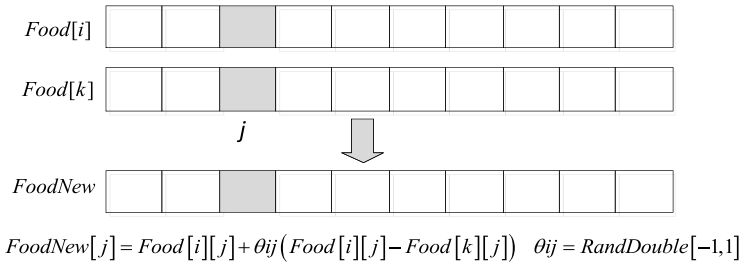

Fig. 5. Example for modifying a food source.

process, the first stage is about the employed bees, in which a number of employed foragers equal to the number of food sources are sent to search for new food sources and to select a better one. The second stage is about the onlooker. We generate a uniform and random food source for each onlooker through the roulette wheel selection method and then send the onlookers to search for a new food source nearby and to select a better one by comparing the fitness. The third stage is about the scouter. If the new food sources generated in the first two stages are worse than the old ones, then both the old ones should be abandoned, and the onlookers begin to search for a new one instead. The algorithm can make a judgment according to the value of *trial*.

We record the current optimal solution in each iteration. The loop continues until the total iterations *maxCycle* are reached, and the optimal solution is finally output.

6.3. Time complexity analysis

We consider the time complexity of ABC in only one iteration. Three steps mainly exist in one generation, namely, the searching of employed bees, onlooker bees, and scouter. During the searching of employed bees, the outside loops are *SN* times, and the inside loops are *limit* times. Thus, the time complexity is $O(SN \times limit)$. The time complexity of the searching of onlooker bees is the same as that of employed bees. In the searching process of the scouter, the time complexity is $O(SN)$.

Thus, the total time complexity of ABC is $O(SN \times limit) + O(SN \times limit) + O(Food\ Number) = O(SN \times limit)$.

6.4. Determining the optimal parameters of ABC

The performance of ABC is closely related to its parameter settings. The key parameters of ABC are *FoodNumber* (equal to *SN*) and *limit*.

FoodNumber should not be significantly large. *FoodNumber* is set to ensure that ABC converges to a good result. When *FoodNumber* is large enough, increasing the population size will not provide considerable benefits. By contrast, *limit* should not be significantly small because a small *limit* will cause ABC to behave similar to random search.

(1) **Determining the optimal *FoodNumber***

FoodNumber varies from 20 to 100. The fitness at a step of 10 is shown in Fig. 6. The *x*-axis represents different *FoodNumbers*, the *y*-axis represents the corresponding fitness, and the different curves represent the fitness of different *VMNum*.

Table 6 lists the detailed experimental data about the fitness value of ABC under different *FoodNumbers*. We use "*" to represent the best situation under different *VMNum*.

When *FoodNumber* is 70, three situations occur (*VMNum* = 20, 80, 100) in which it is the optimum. With the expanding scale, the advantages of *FoodNumber* of 70 become obvious. For the other two situations, the fitness of *FoodNumber* = 70 is also close to the best. We accordingly use 70 as the optimal *FoodNumber*.

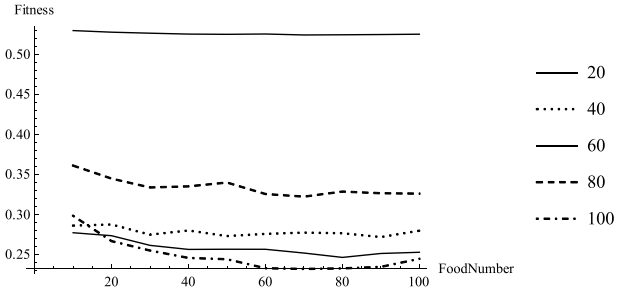

Fig. 6. Fitness value of ABC under different *FoodNumbers*.

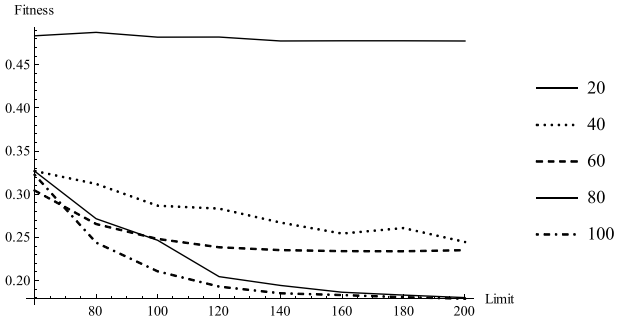

Fig. 7. Fitness value of ABC under different *limit*.

Table 6
Fitness value of ABC under different *FoodNumbers*.

FoodNumber	*VMNum*				
	20	40	60	80	100
10	0.529334	0.285945	0.277035	0.360831	0.298176
20	0.527356	0.287301	0.273294	0.344595	0.266546
30	0.526118	0.274603	0.261257	0.333528	0.254792
40	0.524964	0.27984	0.256188	0.334909	0.245553
50	0.524626	0.272883	0.25644	0.339677	0.243981
60	0.525056	0.275578	0.256416	0.325347	0.23273
70	**0.523851***	0.277177	0.251687	**0.32209***	**0.231864***
80	0.524122	0.276405	**0.246237***	0.328419	0.232515
90	0.524403	**0.271628***	0.251133	0.326323	0.234431
100	0.52485	0.279552	0.252643	0.325862	0.244477

(2) **Determining the optimal *limit***

limit varies from 60 to 200. The fitness at a step of 20 is shown in Fig. 7. The *x*-axis represents the different limits, the *y*-axis represents the corresponding fitness, and the different curves represent the fitness of different VMNum.

limit = 200 is the optimum under almost all different *VMNum*. Thus, *limit* is set to 200.

6.5. Analysis of the principle of ABC

ABC generates a new solution using Eq. (15) with the reference part of parent solutions and a randomly selected solution, and accelerates the current optimal solution convergence, which is

different from GA; GA generates a new solution focused only on the crossover and mutation of parent solutions.

In ABC, the optimal solution is generally not obtained in the initial solution because the initial solution may be replaced with a new one by a scouter. In the stage of the employed bees, each potential solution is developed from the solution set; in the stage of the onlooker, the solution with higher fitness is more likely to be selected as a optimal one; in the stage of the scouter, the potential solutions are handled. The entire process means that ABC can obtain the optimal solution faster and better.

Two mechanisms can be used as bases to ensure the diversity of food sources. (1) A part of the parent solutions should be replaced with the random method to generate new solutions, and such small modifications help to find a better solution. (2) In the stage of the scouter, a new solution should be randomly generated to replace the old one rather than replace a part of the parent solutions. These mechanisms enable ABC to avoid the premature problem.

7. Experiment

7.1. Experimental setup

We use four servers (configuration: AMD Opteron Processor 6136 CPU, 32 GB memory, 500 GB HDD, and Ubuntu Server Version 11.10 operating system) to implement our simulation experiments. The dependency graphs of VMs are generated using BRITE [30], which is a universal network topology generator. The data center topology is set as a tree topology. The parameters used in conducting our simulation experiment are as follows:

(1) *VMNum*.
(2) The number of servers.
(3) The overloaded ratios of VMs.
(4) The interdependencies among the servers that represent various application topologies.
(5) The interdependencies among VMs, the distribution of the traffic demands of the VMs, and the load characteristics of the VMs.
(6) Distribution of the capacities of servers.

7.2. Experimental results

For NetVMM, we first experimentally determine the value of weight coefficients in the optimization function. We then compare the performance in network costs and the running time of four swarm intelligence algorithms, namely, GA, ACO, PSO, and ABC. We give the detailed parameter settings for static migration as follows:

We generate a range of scenarios for our simulations, including varying the number of machines for data center topologies, *VMNum* and the interdependencies among VMs, and the load and capacity characteristics of PMs and VMs. The number of PMs is set to 100, whereas *VMNum* varies from 20 to 120 in a step of 20. The fraction of overloaded VMs is set to 40%. We use the representative data center PM architecture *Tree* to compute the distances among PMs. (See Table 7.)

(1) **Determining weight coefficients α and β**

The first problem to solve is determining the weight coefficients in the objective functions α and β, $\alpha + \beta = 1$. The weight coefficient is a key factor to ensure a good trade-off between communication and migration costs. The setting of weight coefficients should generally make our problem model adaptive to various application scenarios. To ensure scalability and adaptability, we need to make the objective value range as small as possible, i.e., it does not change much under different workloads.

In the experiment, we use GA as the representative algorithm to make *VMNum* change from 20 to 100 at a step of 20 and to test the fitness value under different α (α changes from 0.0 to 1.0 at a step of 0.1). The results are shown in Fig. 8. The *x*-axis represents the change in α, the *y*-axis represents the fitness value, and the different curves show different *VMNum*.

Table 7
Parameters for the simulation of static VM migration.

Variable	Mean	Var	Distribution
Fraction of overloaded VMs			0.4
Size (VM)	0.4	0.2	Uniform
Load (VM)	0.4	0.1	Uniform
W (VM)			BRITE
Capacity (PM)	0.8	0.2	Uniform
Number of VMs			20–120, in steps of 20
Number of PMs			100
PM architecture			*Tree*
Static migration algorithms			GA, ACO, PSO, ABC

Table 8
Fitness value of GA under different weight coefficient α.

α	VMNum				
	20	40	60	80	100
0.0	0.127947	0.0556738	0.0623839	0.0719789	0.0778723
0.1	0.113326	0.0964067	0.0954392	0.102445	0.113421
0.2	0.140574	0.137761	0.1307	0.139701	0.154599
0.3	**0.174551**	**0.169494**	**0.174227**	**0.173352**	**0.187855**
0.4	0.19233	0.192681	0.210393	0.212733	0.222605
0.5	0.229359	0.235865	0.23801	0.258819	0.260628
0.6	0.255474	0.266157	0.271604	0.277824	0.289441
0.7	0.273416	0.295627	0.295616	0.300681	0.322837
0.8	0.296079	0.308649	0.323531	0.31351	0.331305
0.9	0.271482	0.314097	0.347946	0.323851	0.341589
1.0	0.24806	0.30471	0.335979	0.310231	0.33086

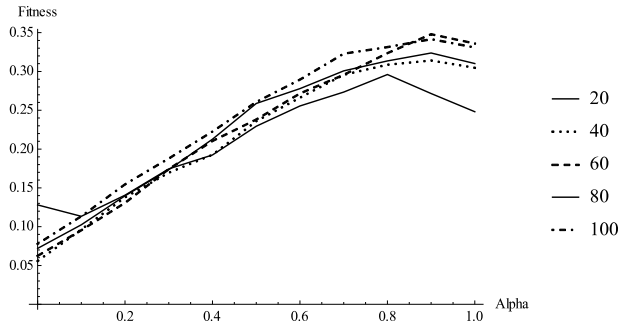

Fig. 8. Fitness value of GA under different weight coefficient α.

Fig. 8 shows that the difference between varied *VMNum* is minimal when α is 0.3. Table 8 lists the detailed experimental information. This table indicates that under $\alpha = 0.3$, the fitness values of different *VMNum* are 0.174551, 0.169494, 0.174227, 0.173352, and 0.187855. The mean difference is minimal. Thus, we set $\alpha = 0.3$ and $\beta = 0.7$ as the weight coefficients for VM migration algorithms.

(2) **Comparing the performance of four swarm intelligence algorithms**

After numerous experiments, we determine the best parameter settings of the four swarm intelligence algorithms, which are shown in Table 9.

We compare the fitness (the *Total_Cost* in figures) and the running time (the *Elapsed_time* in figures) performances of the four algorithms, which are shown in Figs. 9 and 10, respectively.

In Fig. 9, the *x*-axis denotes the change in *VMNum*, and the *y*-axis denotes the fitness values of the algorithms. In each group of bars, the fitness values of GA, ACO, PSO, and ABC are shown from the left to the right. Fig. 9 illustrates that the performance of GA is always stable. Table 10 lists the experimental data in detail.

Table 9
Parameter settings of ACO, GA, PSO, and ABC.

ACO		GA		PSO		ABC	
Parameters	Value	Parameters	Value	Parameters	Value	Parameters	Value
ρ	0.2	*Crossover rate*	0.6	ω	0.9-0.4, decrease linearly with iterations	*SN*	70
p_{best}	0.05	*Mutation rate*	0.1	$C1$	1	*limit*	200
α	1	*PopSize*	100	$C2$	1		
β	0	Crossover type	Typically two point crossover	Neighborhood topology of local-version PSO	Full		
nAnts		Mutation type	Bit-flop mutation	Number of particles	40		
Iterations	5000	*nIterations*	5000	*nIterations*	5000	*nIterations*	5000

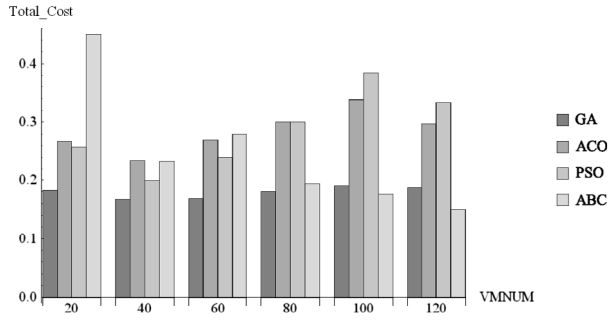

Fig. 9. Fitness values of the four swarm intelligence algorithms.

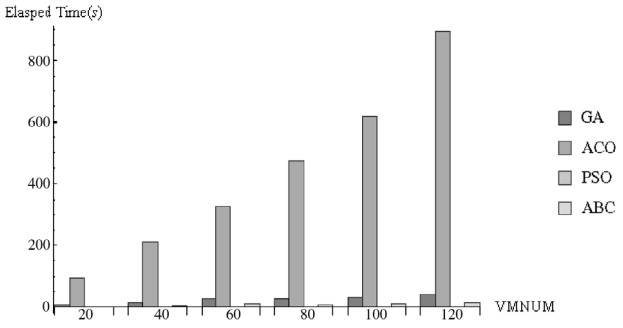

Fig. 10. Running time of the four swarm intelligence algorithms.

Table 10
Fitness values of the four swarm intelligence algorithms.

VMNum	Algorithm			
	GA	ACO	PSO	ABC
20	**0.183147**	0.26774	0.258243	0.449951
40	**0.167335**	0.233621	0.199013	0.232834
60	**0.168566**	0.270189	0.240831	0.279825
80	**0.180647**	0.300764	0.300894	0.193734
100	0.190051	0.338123	0.384781	**0.176236**
120	0.187259	0.29729	0.333197	**0.150247**

When *VMNum* = 20, 40, 60, 80, GA has the smallest fitness value. With an increase in the problem size, the fitness value of ABC increases. When *VMNum* = 100 and 120, ABC has the best fitness value. Therefore, ABC has a better performance than GA when the problem instances are large. Nonetheless, GA is stable in solving the NetVMM problem under different problem instances.

Fig. 10 shows the running time of the four swarm intelligence algorithms under different *VMNum*. PSO has the fastest running time, followed by ABC, GA, and ACO. However, ACO and PSO perform inefficiently in fitness value. Having a fast running time with an inefficient network performance is meaningless. Fig. 11 shows the running time of GA and ABC. The running time of ABC is only approximately half that of GA. Thus, ABC has another advantage over GA.

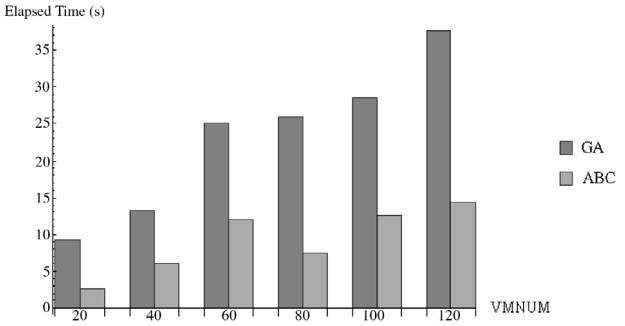

Fig. 11. Running time of GA and ABC.

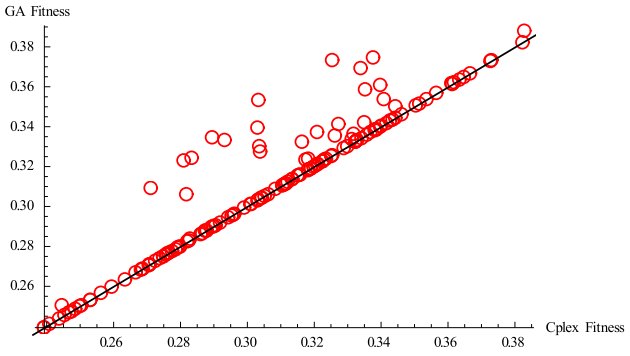

Fig. 12. Comparison of GA with the optimal solution.

(3) **Comparison with CPLEX**

CPLEX software, which can be used to calculate the optimal solution for linear programming, quadratic assignment (quadratically constrained programming), and mixed plastic planning (mixed integer programming), is a software package developed by IBM Scientific Computing [4].

We conduct another experiment to determine the closeness of our swarm intelligence algorithms with optimal solutions. The optimal solutions are computed by CPLEX. Considering that CPLEX supports a maximum of 500 parameters for large-scale problems (otherwise, it will cause memory overflow), we conduct an experiment with small-scale problems. *VMNum* and the number of PM are set to 10. Results are shown in Figs. 12–15.

Figs. 12–15 show the comparison of the fitness of our swarm intelligence algorithms with the optimal solutions returned by CPLEX. The dots in these figures denote the fitness of the swarm intelligence algorithms, and the line $y = x$ in the figure is the benchmark of CPLEX. The closer these dots are distributed near $y = x$, the closer the swarm intelligence algorithms are to the optimal solutions.

GA is the closest to the optimal solutions under the small-problem instance. ACO ranks as the second, PSO is the third, and ABC is loosely distributed far away from the line $y = x$. We also compare the standard deviations of the results of the four

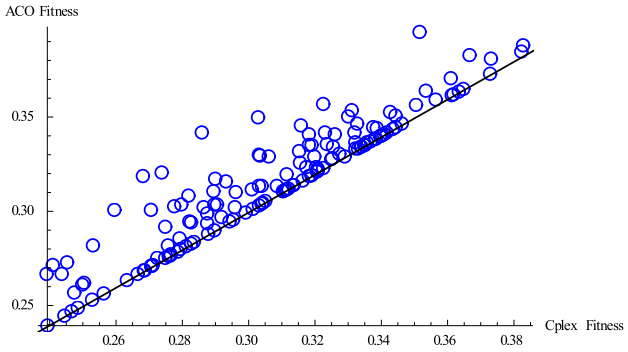

Fig. 13. Comparison of ACO with the optimal solution.

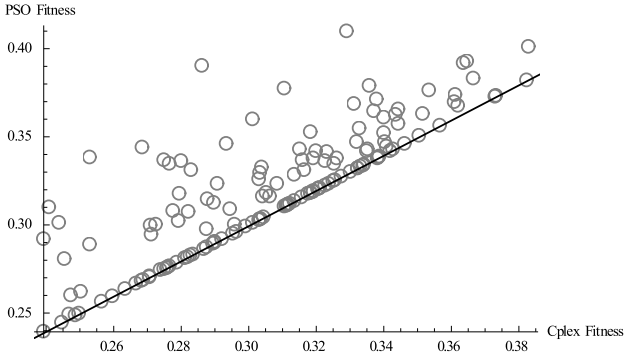

Fig. 14. Comparison of PSO with the optimal solution.

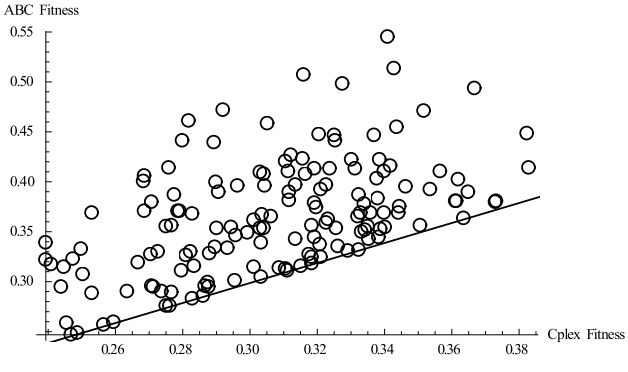

Fig. 15. Comparison of ABC with the optimal solution.

algorithms, as presented in Table 11. This comparison validates our conclusion.

We list another example for VM of 20 SM of 10. The standard deviations of the results of the four algorithms are shown in Table 12, which also validate our conclusion. The results are similar to the preceding results.

8. Conclusion

This study considers the NetVMM in an overcommitted cloud and transforms it into an NP-complete problem. This study aims to minimize network traffic costs by considering the inherent dependencies among VMs that comprise a multi-tier application and the underlying topology of PMs, as well as to ensure a good trade-off between network communication and VM migration costs. The mechanism that the swarm intelligence algorithms aim to find is an approximate optimal solution through repeated iterations to make it a good solution for the VM migration problem. In this study, GA and ABC are adopted to solve the NetVMM problem. Experimental results show that GA has a better network cost when VM instances are small. When the problem size increases, ABC is advantageous to GA. The running time of ABC is also approximately only half that of GA, which is another advantage. This study is the first to use ABC to solve the NetVMM problem.

Table 11
Standard deviations of the four algorithms with CPLEX (VM10, SM10).

GA	ACO	PSO	ABC
0.0118654	0.0146607	0.0146607	0.075507

Table 12
Standard deviations of the four algorithms with CPLEX (VM20, SM10).

GA	ACO	PSO	ABC
0.0204988	0.0235531	0.0286359	0.0524387

Acknowledgments

This work was supported by the National Natural Science Foundation of China (NSFC) under Grant Nos. 61173145 and 61472108, and the Doctoral Program of Higher Education of China (RFDP) under Grant No. 20132302110037.

References

[1] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, N. McKeown, ElasticTree: Saving energy in data center networks, in: Proc of the 7th UXENIX Symposium Networked Systems Design and Implementation, NSDI'10, vol. 3, 2010, pp. 249–264.
[2] A. Beloglazov, J.H. Abawajy, R. Buyya, Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing, Future Gener. Comput. Syst. 28 (5) (2012) 755–768.
[3] J.L. Berral, I. Goiri, R. Nou, F. Julià, J. Guitart, R. Gavaldà, J. Torres, Towards energy-aware scheduling in data centers using machine learning, in: Proc. of the 1st International Conference on Energy-Efficient Computing and Networking, e-Energy'10, 2010, pp. 215–224.
[4] M. Hsu, T.E. Cheatham, Rule execution in CPLEX: A persistent objectbase, in: Advances in Object-Oriented Database Systems, in: Lecture Notes in Computer Science, vol. 334, 1988, pp. 150–155.
[5] J. Chou, J. Kim, D. Rotem, Energy-aware scheduling in disk storage systems, in: Proc. of the 31st International Conference on Distributed Computing Systems, ICDCS'11, 2011, pp. 423–433.
[6] S. Srikantaiah, A. Kansal, F. Zhao, Energy aware consolidation for cloud computing, in: Proc. of the 2008 Conference on Power Aware Computing and Systems, HotPower'08, USENIX Association, 2008, 1–5.
[7] A. Verma, P. Ahuja, A. Neogi, PMapper: Power and migration cost aware application placement in virtualized systems, in: Proc. of the 9th ACM/IFIP/USENIX International Conference on Middleware, Middleware'08, 2008, pp. 243–264.
[8] L. Lefèvre, A. Orgerie, Designing and evaluating an energy efficient cloud, J. Supercomput. 51 (3) (2010) 352–373.
[9] H. Liu, C. Xu, H. Jin, J. Gong, X. Liao, Performance and energy modeling for live migration of virtual machines, in: Proc. of the 20th International Symposium on High Performance Distributed Computing, HPDC'11, 2011, pp. 171–182.
[10] V. Mann, A. Kumar, P. Dutta, S. Kalyanaraman, VMFlow: Leveraging VM mobility to reduce network power costs in data centers, in: Proc. of the 10th International IFIP TC 6 Conference on Networking—Volume Part I, Networking'11, 2011, pp. 198–211.
[11] M. Zhani, Q. Zhang, G. Simon, R. Routaba, VDC Planner: Dynamic migration-aware virtual data center embedding for clouds, in: Proc. the 13th IFIP/IEEE International Symposium on Integrated Network Management, IM'13, 2013, pp. 18–25.
[12] X. Wen, K. Chen, Y. Chen, Y. Liu, Y. Xia, C. Hu, VirtualKnotter: Online virtual machine shuffling for congestion resolving in virtualized datacenter, in: Proc. of the 32nd International Conference on Distributed Computing Systems, ICDCS'12, 2012, pp. 12–21.
[13] D. Kliazovich, P. Bouvry, S.U. Khan, DENS: Data center energy-efficient network-aware scheduling, Cluster Comput. 16 (1) (2013) 65–75.
[14] A.G. Greenberg, J.R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D.A. Maltz, P. Patel, S. Sengupta, VL2: A scalable and flexible data center network, in: Proc. the 29th ACM Special Interest Group on Data Communication, SIGCOMM'09, 2009, pp. 51–62.
[15] X. Meng, V. Pappas, L. Zhang, Improving the scalability of data center networks with traffic-aware virtual machine placement, in: Proc. of the 29th Conference on Computer Communications, INFOCOM'10, 2010, pp. 1154–1162.

[16] D. Jayasinghe, C. Pu, T. Eilam, M. Steinder, I. Whalley, E.C. Snible, Improving performance and availability of services hosted on IaaS clouds with structural constraint-aware virtual machine placement, in: Proc. the 8th IEEE International Conference on Services Computing, SCC'11, 2011, pp. 72–79.
[17] V. Shrivastava, P. Zerfos, K. Lee, H. Jamjoom, Y. Liu, S. Banerjee, Application-aware virtual machine migration in data centers, in: Proc. of the 30th Conference on Computer Communications, INFOCOM'11, 2011, pp. 66–70.
[18] X. Zhang, Z. Shae, S. Zheng, H. Jamjoom, Virtual machine migration in an over-committed cloud, in: Proc. of the 13th Network Operations and Management Symposium, NOMS'12, 2012, pp. 196–203.
[19] J. Chen, W. Liu, J. Song, Network performance-aware virtual machine migration in data centers, in: Proc. of the Third International Conference on Cloud Computing, GRIDs, Cloud Computing 2012, 2012, pp. 65–71.
[20] S. Ghorbani, M. Caesar, Walk the Line: Consistent network updates with bandwidth guarantees, in: Proc. of the First Workshop on Hot Topics in Software Defined Networks, HotSDN'12, 2012, pp. 67–72.
[21] T.D. Braun, H.J. Siegel, N. Beck, L. Bölöni, M. Maheswaran, A.I. Reuther, J.P. Robertson, M.D. Theys, B. Yao, D.A. Hensgen, R.F. Freund, A comparison of eleven static heuristics for mapping a class of independent tasks onto heterogeneous distributed computing systems, J. Parallel Distrib. Comput. 61 (6) (2001) 810–837.
[22] H. Chen, A.M.K. Cheng, Y. Kuo, Assigning real-time tasks to heterogeneous processors by applying ant colony optimization, J. Parallel Distrib. Comput. 71 (1) (2011) 132–142.
[23] S. Agrawal, S.K. Bose, S. Sundarrajan, Grouping genetic algorithm for solving the server consolidation problem with conflicts, in: Proc. of the First ACM/SIGEVO Summit on Genetic and Evolutionary Computation, GEC Summit'09, 2009, pp. 1–8.
[24] W. Zhang, H. Xie, R. Hsu, Automatic memory control of multiple virtual machines on a consolidated server, IEEE Trans. Cloud Comput. (99) 1–14. http://dx.doi.org/10.1109/TCC.2014.2378794.
[25] W. Zhang, H. He, G. Chen, J. Sun, Multiple virtual machines resource scheduling for cloud computing, Appl. Math. 7 (5) (2013) 2089–2096.
[26] M.J. Magazine, M.-S. Chern, A note on approximation schemes for multidimensional knapsack problems, Math. Oper. Res. 9 (2) (1984) 244–247.
[27] K.A.D. Jong, W.M. Spears, An analysis of the interacting roles of population size and crossover in genetic algorithms, in: Parallel Problem Solving from Nature, in: Lecture Notes in Computer Science, vol. 496, 1991, pp. 38–47.
[28] K. DeJong, Learning with genetic algorithms: An overview, Mach. Learn. 3 (2–3) (1988) 121–138.
[29] D. Karaboga, B. Akay, A comparative study of artificial bee colony algorithm, Appl. Math. Comput. 214 (1) (2009) 108–132.
[30] A. Medina, A. Lakhina, I. Matta, J.W. Byers, BRITE: An approach to universal topology generation, in: Proc. of the Ninth International Symposium on Analysis and Simulation of Computer and Telecommunication Systems, MASCOTS'01, 2001, pp. 346–353.

Weizhe Zhang is a professor in the School of Computer Science and Technology at Harbin Institute of Technology, China. He has been a visiting scholar in the Department of Computer Science at University of Illinois at Urbana–Champaign and University of Houston, USA. His research interests are primarily in parallel computing, distributed computing, and cloud computing. He has published more than 100 academic papers in journals, books, and conference proceedings. He is a member of the IEEE.

Shuo Han has received a bachelor's degree from the School of Computer Science and Technology at Harbin Institute of Technology, China in 2014. She is currently a master's candidate at Harbin Institute of Technology. Her research interest involves virtualization techniques for cloud computing.

Hui He is currently an associate professor of network security center in the Department of Computer Science, China. She received the Ph.D. from department of computer science at the Harbin Institute of Technology, China. Her research interests are mainly focused on distributed computing, IoT and big data analysis. Contact her at hehui@hit.edu.cn.

Huixiang Chen was born in HeNan in 1989. He received the B.S. degree from the Department of Computer and Communication Engineering, Jilin University, Jilin, China, in 2011. He received the M.S. degree from the Department of Computer Science and Technology, Harbin Institute of Technology, Harbin, China, in 2013. He is currently Ph.D. student of the Department of Computer Science, University of Florida, Florida, USA. His research interests include parallel computing, and cloud computing.