A Group Genetic Algorithm for Resource Allocation in Container-based Clouds

Boxiong Tan, Hui Ma, and Mei Yi

Victoria University of Wellington, New Zealand {Boxiong.Tan, Hui.Ma, Mei.Yi}@ecs.vuw.ac.nz

Abstract. Containers have gain popularity because they support fast development and deployment of cloud-native software such as micro-services and server-less applications. Additionally, containers have low overhead, hence they save resources in cloud data centers. However, the difficulty of the *Resource Allocation in Container-based clouds (RAC)* is far beyond Virtual Machine (VM)-based clouds. The allocation task selects heterogeneous VMs to host containers and consolidate VMs to Physical Machines (PMs) simultaneously. Due to the high complexity, existing approaches use simple rule-based heuristics and meta-heuristics to solve the *RAC* problem. They either prone to stuck at local optima or have inherent defects in their indirect representation. To address these issues, we propose a novel *group genetic algorithm (GGA)* with direct representation and problem-specific operators. This design has shown significantly better performance than the state-of-the-art algorithms in with a wide range of test datasets.

Keywords: cloud resource allocation · container placement · energy consumption · group genetic algorithm.

1 Introduction

Container-based clouds [15] have quickly become a new trend in cloud computing. Compared to Virtual Machines (VMs), containers (e.g. docker) cause much fewer overheads. This feature is critical for modern cloud-native applications, such as microservices and serverless applications, as they are developed in a decoupling and scalable manner. Cloud providers also welcome containerization because resource utilization can be improved by sharing a VM with multiple applications. Consequently, higher utilization of resources leads to the lower energy consumption of cloud data centers, therefore, containers can help to achieve a green cloud [4].

Cloud providers apply server consolidation [22] strategies in resource allocation to improve the utilization of cloud resources. Server consolidation strategies aim to allocate applications to a minimum number of Physical Machines (PMs), to reduce energy consumption. In container-based clouds, it is much difficult than in VM-based clouds because of the higher granularity of the allocation problem. Server consolidation in VM-based clouds involves one level of allocation, i.e. a set of VMs is allocated to PMs directly while container-based clouds involve two levels of allocation, i.e. a set of containers is allocated to a set of VMs with

various types, and the VMs are allocated PMs. In the remaining of this paper, we use *Resource Allocation in Container-based clouds (RAC)* to represent the consolidation problem. In terms of difficulty, the two levels of allocation are both vector bin packing problems which are NP-hard [23]. Moreover, resource allocation in the first level, e.g., VM type selection, impacts the resource allocation in the second level.

Since it is impossible to find the optimal solution for a large scale *RAC* problem (e.g. over 1000 containers), existing studies mainly apply rule-based heuristics [7, 12, 15, 23], and meta-heuristics algorithms [2, 5, 20] to find near-optimal solutions. Rule-based heuristics are greedy so they prone to stuck at local optimal solutions and they perform differently when facing various settings of VM types from multiple cloud providers. The meta-heuristics are promising algorithms. However, the current research either focuses on allocating containers directly to PMs problem or uses indirect representation which is inefficient in the searching process.

GGA was proposed by Falkenauer [3] and inspired many studies in solving the VM allocation problem [11, 21]. Different from the standard GA, *GGA* applies a variable length of chromosome and domain-specific genetic operators such as inversion and rearrangement. *GGA* is designed for bin packing problem and uses a direct representation which avoids a decoding process. However, *GGAs* [3, 17] can only solve one-level problems.

This research aims at proposing a novel *Group GA (GGA)* for the *RAC* problem to minimize the energy consumption. The proposed *GGA* approach provides the functionality of selecting VM types. Also, it has a direct representation and problem-specific operators to address the limitations of the dual-chromosome GA approach. To achieve our aim, we set up the following objectives:

1. To propose a new representation for the *RAC* problem,
2. To develop new genetic operators including gene-level crossover, unpack, rearrangement, and merge.
3. To evaluate our proposed approach by comparing it with the state-of-the-art algorithms: Rule-based (FF&BF/FF) approach [23] and two variations of dual-chromosome GA [20] approaches.

The paper is organized as follows. Section 2 gives a background of our methodology and discusses related studies of the *RAC* problem. Section 3 presents the model of the problem. Then, section 4 describes the proposed *GGA* approach. Section 5 illustrates the experiment design, results, and analysis. Section 6 summarizes the contributions and discusses the future works.

2 Related Work and Background

This section first reviews related works of the resource allocation in container-based clouds. Then, we provide a brief background of *GGA* [3].

2.1 Related Works

Current studies solve the *RAC* problem with two types of method, rule-based approaches, and meta-heuristics approaches. Piraghaj [15], Kaur [7], Mann [13], Liu [10] and Zhang [23] treat the problem as a dynamic problem and propose AnyFit-based (e.g. First Fit, Best Fit) approaches to solve the problem. The proposed rules evaluate the candidate VMs and VM types to decide which VM to choose or which VM type to create. Overall, from the problem's perspective, as Wolke et al. [22] suggested, dynamic approaches are useful in some scenarios such as container migration and inferior in other scenarios such as initial container allocation. From the methods' perspective, the rules have a poor generality. Their performance varies when applying them to different settings of VM types (see Section 5.4). Another drawback is that these greedy rules are easily stuck at local optimal solutions.

A few meta-heuristics have been proposed, but they are either focus on one-level allocation problem such as [5, 9], or uses an indirect representation [2, 20]. Guerrero et al. [5] propose an NSGA-II-based approach for a four-objective allocation problem. Lin et al. [9] propose an ant colony algorithm-based approach for the problem. In their models, containers are allocated directly to PMs without considering VMs. Tan et al. [2,20] propose two meta-heuristic approaches for the *RAC* problem, an NSGA-II-based and a dual-chromosome GA (DGA) approach. These approaches use indirect representations and they require a decoding process to interpret the representation to a solution. Overall, these algorithms search in the genotype space.

The current meta-heuristics have two shortcomings. The first drawback is that they [5, 9] only consider the one-level structure which inherently leads to local optimal solutions. The second drawback is that the decoding process of [20] can easily break the solutions (good combination of containers and VMs) from the previous generation. Therefore, it is hard to perform a directed search. As a consequence, the algorithms with indirect representation cannot find local optimal solutions efficiently.

Therefore, because of these drawbacks in the literature, we propose a meta-heuristic with a direct representation to solve the two-level *RAC* problem. The next section discusses the background of the *GGA* and explains how it can be adapted to our problem and meets our goal.

2.2 Group Genetic Algorithm (GGA)

GGA was proposed by Falkenauer [3] to solve the bin packing problem. *GGA* overcomes a major defect, the redundant encoding problem, in the ordering GA [16]. The ordering GA uses an encoded representation and the decoding process highly relies on items rather than the numbering of groups. For example, using two letters A and B to represent distinct groups, AAB and BBA are two solutions. However, in terms of grouping, these two solutions have the same meaning – the first two items are in the same group and the third item is in another group. To solve the redundant problem, *GGA* proposes a variable-length

representation. The new crossover, mutation, and inversion operators directly operate on groups instead of items. Later on, Quiroz-Castellanos [17] embeds heuristics into the algorithm to speed up the search procedures.

GGA has been successfully applied to solve many bin packing problems such as ordering batch problems in warehouse [8], VM placement problem [6,11], and assembly line balancing problem [18]. However, it has not to be used to solve any two-level vector bin packing problems. Our *RAC* problem is a two-level vector bin packing problem. It is promising to adopt *GGA*'s framework and propose problem-specific operators to solve our problem.

3 Problem Model

Resource Allocation in Container-based clouds problem (RAC) is a task of allocating a set of containers to a set of VMs with various types, then allocating the created VMs to a set of PMs. VM selection chooses an existing VM to allocate a container. VM creation selects a type of VM, creates a VM with the selected type and allocates the container to the new VM. The types of VM are defined by cloud providers. PM selection chooses an existing PM to allocate the new VM. If there is no available PM, a new PM will be created and the data center automatically allocates the new VM to the new PM. Since the PMs are homogeneous, no decision is needed for PM creation.

In the static setting of *RAC* problem, a set of containers $\mathcal{C} = \{c_1, \ldots, c_n\}$ arrives to the cloud to be allocated. Each container c_i has a CPU occupation $\zeta^{cpu}(c_i)$, a memory occupation $\zeta^{mem}(c_i)$. There is a set of VM types $\Gamma = \{\tau_1, \ldots, \tau_m\}$ that can be selected to allocate the containers. Each VM type τ_j has a CPU capacity $\Omega^{cpu}(\tau_j)$ and a memory capacity $\Omega^{mem}(\tau_j)$. Also, it has a CPU overhead $\pi^{cpu}(\tau_j)$ and memory overhead $\pi^{mem}(\tau_j)$, indicating the CPU and memory occupation for creating a new VM of that type. There is an unlimited set of PMs $\mathcal{P} = \{p_1, \ldots, \}$ for allocating the created VMs. Each PM p_k has a CPU capacity $\Omega^{cpu}(p_k)$ and a memory capacity $\Omega^{mem}(p_k)$.

The static *RAC* problem is subject to the following constraints:

1. Each container is allocated to one VM.
2. Each created VM is allocated to one PM.
3. For each created VM, the total CPU and memory occupations of the containers allocated to that VM does not exceed the corresponding VM capacity.
4. For each PM, the sum of the CPU and memory capacities of the VMs allocated on the PM does not exceed the corresponding PM's capacity.

The energy consumption is calculated as follows:

$$E = \sum_{k=1}^{K} E_k, \tag{1}$$

where E_k is the energy consumption of the kth PM (K is the number of PM used).

E_k is calculated as follows:

$$E_k = E_k^{idle} + (E_k^{full} - E_k^{idle}) \cdot \mu_k^{cpu}, \tag{2}$$

where E_k^{idle} and E_k^{full} indicate the energy consumption of the kth PM per time unit if it is idle and fully loaded, respectively. μ_k^{cpu} indicates the CPU utilization level of the kth PM. μ_k^{cpu} is calculated as follows.

$$\mu_k^{cpu} = \frac{\sum_{l=1}^{L} \left(\sum_{j=1}^{m} \pi^{cpu}(\tau_j) \cdot z_{jl} + \sum_{i=1}^{n} \Omega^{cpu}(c_i) \cdot x_{il} \right) \cdot y_{lk}}{\Omega^{cpu}(p_k)}, \tag{3}$$

where x_{il}, y_{lk} and z_{jl} are binary decision variables, and L is the number of created VMs. x_{il} takes 1 if c_i is allocated to the lth created VM, and 0 otherwise. y_{lk} takes 1 if the lth created VM is allocated to the kth PM, and 0 otherwise. z_{jl} takes 1 if the lth created VM is of type j, and 0 otherwise.

The static RAC problem is to find resource allocation with minimal overall energy consumption as shown as follows.

$$\min \ \sum_{k=1}^{K} E_k, \tag{4}$$

$$s.t. \ \sum_{l=1}^{L} x_{il} = 1, \ \forall \ i = 1, \ldots, n, \tag{5}$$

$$\sum_{k=1}^{K} y_{lk} = 1, \ \forall \ l = 1, \ldots, L, \tag{6}$$

$$\sum_{j=1}^{m} z_{jl} = 1, , \ \forall \ l = 1, \ldots, L, \tag{7}$$

$$\sum_{i=1}^{n} \zeta^{res}(c_i) x_{il} \leq \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl}, \quad \forall \ l = 1, \ldots, L, \ res \in \{cpu, mem\}, \tag{8}$$

$$\sum_{l=1}^{L} \sum_{j=1}^{m} \Omega^{res}(\tau_j) z_{jl} \leq \Omega^{res}(p_k), \quad \forall \ k = 1, \ldots, K, \ res \in \{cpu, mem\}, \tag{9}$$

$$x_{il}, y_{lk}, z_{jl} \in \{0, 1\}, \tag{10}$$

where constraints (5) and (6) indicate that each container (or new created VM) is allocated to exactly one created VM (or PM). Constraint (7) indicates that each created VM must belong to a type. Constraint (8) implies that the total occupation of all the containers allocated to each created VM does not exceed its corresponding capacity. Constraint (9) indicates that the total capacity of the

created VMs allocated to each PM does not exceed its corresponding capacity. Constraint (10) defines the domain of the decision variables.

The energy calculation of the model is used in the fitness function of our proposed algorithm. The constraints of the model are used in the algorithm to ensure the solutions are valid.

4 The Proposed Group GA for the RAC Problem

This section describes our *GGA* approach for the *RAC* problem which includes a group representation and three problem specific operators.

4.1 Overall Framework

Algorithm 1: Group genetic algorithm for the *RAC* problem

```
   Input  : a set of containers, a set of VM types, a list of PMs,
   Output: an allocation of containers
 1 population ← Initiailization;
 2 gen ← 0 ;
 3 for gen does not reach the maximum generation do
 4 |   fitness evaluation(population);
 5 |   new population ← elitism(population);
 6 |   while has not fill the new population do
 7 |   |   parents ← tournament selection(population);
 8 |   |   children ← gene-level crossover(parents);
 9 |   |   unpack(children);
10 |   |   merge(children);
11 |   |   add children to the new population
12 |   end
13 |   gen ← gen + 1;
14 end
15 return an allocation of containers;
```

The algorithm (see Algorithm 1) starts with the initialization of a population. The individual is represented as a list of PMs. Then, the algorithm enters a loop of evolution where each loop is called a generation. In each generation, individuals are evaluated with a fitness function (Eq.(1)). Then, the top individuals are preserved and copied to the new population with Elitism [1]. Tournament selection [14] is used to direct the population to the high-fitness region. Then, we proposed three problem-specific operators, gene-wise crossover, unpack, and merge. These operators modify the individuals so that they can perform an effective search in the solution space.

4.2 Representation

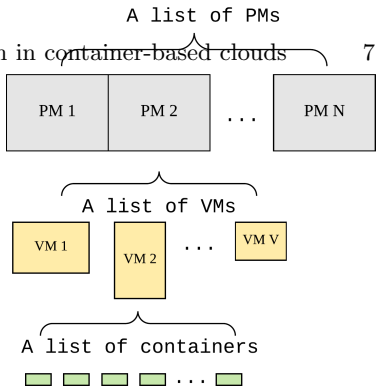


Fig. 1: Representation

The representation of an individual (see Figure 1) is a complete solution for a *RAC* problem. The individual consists of a list of PMs. Each PM consists of a list of VMs and each VM has a list of containers. This representation can be directly evaluated without using any decoding process. More importantly, the direct representation can be modified by heuristics at a specific point, e.g. switch two containers' allocation, without changing the structure of the entire solution. Therefore, the disadvantage of indirect representation in *dual-chromosome GA* [20] can be avoided.

4.3 Initialization

The design of initialization aims at producing a diverse population of solutions. For each individual, we first randomly generate a permutation of containers. Then, we allocate containers to VMs using the First-Fit heuristic. If there is no VM available, we create a VM with a random type. Lastly, a list of VMs is allocated to PMs with First-Fit. This representation ensures a diverse combination of containers and VMs. It also locates the solutions in a relatively high-quality region with First Fit instead of Next Fit. This is because Next Fit does not guarantee a VM or a PM is filled while First Fit guarantees that. Therefore, the average quality obtained by First Fit is much better than Next Fit.

4.4 Gene-level Crossover

To inherit the useful parts from parents, one must define what is a "good gene". In the bin packing problem, a good gene is at bins' level where well-filled bins can lead to fewer bins [17]. Similarly, highly utilized PMs could lead to fewer PMs in the allocation problem. Therefore, our good gene is defined as a PM with high utilization. In our case, we apply the crossover twice according to the utilization of CPU and memory respectively and generate two children.

The gene-level crossover preserves the highly utilized PMs from both parents. In the beginning, we sort the PMs in both parents according to PMs' utilization of CPU or memory in descending order. Then, the crossover compares the PMs from two parents in pairwise (see Fig.2). The winner's PM of the pair will be preserved. Preservation includes three steps. First, the crossover copies the VMs combination inside the PM including the types and number of VMs. Second, the crossover checks whether a container from the original VM has been allocated in the previous PMs. If the container has been allocated, then the container will not be allocated again. In the end, some containers may not be allocated to PMs. They are called *free containers*. These free containers are reallocated with an operator called *rearrangement* which will be introduced in the next section. After all the containers have been allocated, empty PMs and VMs are removed from an individual.

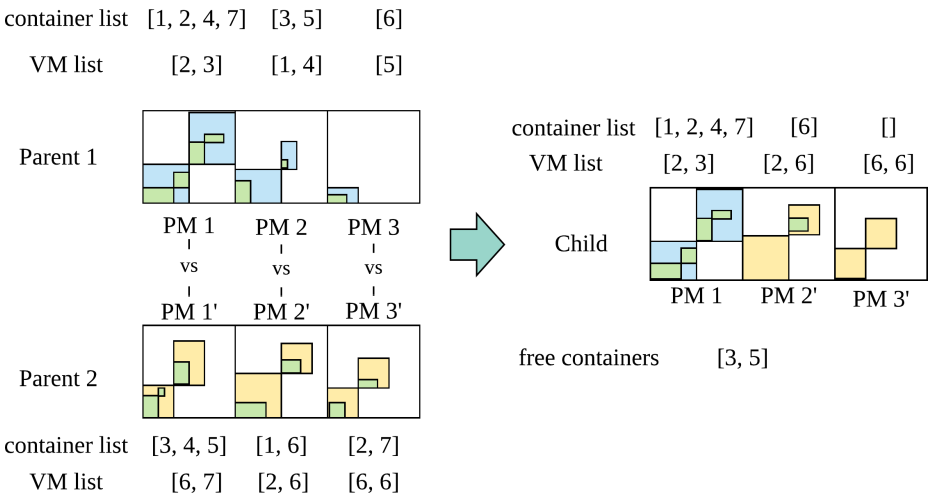


Fig. 2: Gene-level crossover

Fig. 2 shows an example of the gene-level crossover. We first sort the PMs from parents according to their CPU utilization. Then, we compare PMs and preserve the structure of *PM 1*, *PM 2'*, and *PM 3'*. The containers in *PM 1* are preserved while the duplicated containers in *PM 2'* and *PM 3'* are removed. In the end, containers 3 and 5 become free containers and they will be allocated to these PMs using the *rearrangement* operator.

4.5 Rearrangement

Rearrangement inserts free items to bins. In the beginning (see Alg 2), we sort the containers according to the product of their normalized resources (see Eq.11) in ascending order. Then, we check that in each VM, whether the smallest two containers can be replaced by the target container. If so, we replace the small containers with the target container. Otherwise, check the next VM. After replacing, we have two smaller containers need to be allocated. At this point, we apply *First-Fit (FF) & Random Creation (RC) / First-Fit (FF)* heuristics to allocate them. The *FF&RC/FF* means, we first use FF to allocate containers to VMs. If no VM is available, we randomly create a new VM to allocate containers and use FF to allocate the new VM to PMs.

Our rearrangement operator is inspired by [17] to avoid the drawback of First Fit (FF) and further improve the structure of a VM. In the bin packing problem, FF-based approaches [3, 17] have been widely used. However, a simple FF-based approach cannot change the existing structure of a bin. Hence, the replacement heuristic is developed. The core idea of the replacement heuristic is that the smaller items are easier to allocate. Therefore, if we can replace a big container with smaller ones, which can be easily allocated to existing VMs without creating a new VM.

$$R = \frac{\zeta^{cpu}(c_i)}{\Omega^{cpu}(p_k)} \cdot \frac{\zeta^{mem}(c_i)}{\Omega^{mem}(p_k)} \tag{11}$$

Algorithm 2: Rearrangement operator

```
Input  : a target container, a list of PMs,
Output: a list of PMs
Sort the containers in all VMs according to Eq.11 in ascending order;
for each VM do
    if the two smallest containers in each VM can be replaced by the target
     container then
        Replace two containers with the target VM;
        Allocate two containers with FF&RC/FF;
        return a list of PMs;
    end
end
Allocate the target container with FF&RC/FF;
return a list of PMs;
```

4.6 Unpack

Unpack operator eliminates low-utilized PMs and reallocates their containers. This operator prevents premature convergence and introduces new gene component into the current population.

The operator has two steps. First, it calculates the probability of unpacking a PM according to Eq.(12). The PM with high CPU utilization has a smaller chance to be unpacked. Second, it unpacks PMs in a roulette wheel style. After unpacking, the free containers are reallocated with the *rearrangement* operator.

$$probability = \frac{1 - \Omega^{cpu}(p_k)}{\sum_{k=1}^{K} 1 - \Omega^{cpu}(p_k)} \tag{12}$$

The unpack operator is adaptive with the evolution process. In the beginning, the average utilization of PMs is low, therefore, more PMs are unpacked. As the population evolved, high utilized PMs move to the head of an individual and have a low chance to be unpacked. Therefore, the good genes are preserved and new genes are introduced by the *rearrangement* operator.

4.7 Merge

The merge operator replaces small VMs with a bigger one to reduce the free resources in PMs. Free resources represent the resources that have not been allocated to any VMs. The merge operator can improve the utilization of PM by reducing the free resources in PMs as well as the overheads from VMs.

Merge operators have two alternative functionalities, merge and enlarge. In the first one, it goes through all the PMs and checks whether the two smallest VMs can be replaced by a larger VM type. If it is possible, all the containers are migrated from these two small VMs to the new larger VM and the small VMs are removed. If we cannot replace two VMs with a larger one, we attempt to replace the smallest VM with a larger one. The large VM type is also selected randomly.

5 Experiment

The overall goal of the experiment is to test the performance of our proposed *GGA* in terms of energy consumption. We conduct experiments on a real-world dataset and compare the results with three benchmark algorithms (a rule-based approach *FF&BF/FF* and two variations of the *dual-chromosome GA*). Then, we analyze the performance of these approaches and explain the pros and cons of them. Details are shown below.

5.1 Dataset and Test instance

We design 8 test instances (see Table 1) which allocates an increasing number of containers (from 200 to 1500) in two sets of VM types. We use a real-world application trace (AuverGrid trace [19]) as the resource requirements of containers. To generate the containers' resource requirements, we select the first 400,000 lines of the trace from the original datasets. Then we filtered the trace to exclude the containers that require more resources than the largest VM. The last step randomly samples resource requirement and use them as the containers.

For the settings of PMs and VMs, we assume homogeneous PMs which have 8 cores and the total capacity is [13200 MHz, 16000 MB]. The maximum energy consumption for the PM is set to 540 KWh. This setting has been used in [12]. We design two sets of VM types (see Table 2), a real-world VMs (20 types from Amazon EC2) and a synthetic set of VMs (10 types). The real-world VM types are proportional whereases the synthetic ones are random. The CPU and memory of synthetic VM types are sampled from [0, 3300 MHz] and [0, 4000 MB] representing the capacity of one core.

Table 1: Test instances

instance	VM types	number of containers	instance	VM types	number of containers
1	synthetic VM types	200	5	real-world VM types	200
2	synthetic VM types	500	6	real-world VM types	500
3	synthetic VM types	1000	7	real-world VM types	1000
4	synthetic VM types	1500	8	real-world VM types	1500

Table 2: VM types

real world VM types							
VM types	[CPU, Memory]	VM types	[CPU, Memory]	VM types	[CPU, Memory]	VM types	[CPU, Memory]
1	[206.25, 250]	6	[412.5, 1000]	11	[825, 2000]	16	[825, 1875]
2	[412.5, 500]	7	[825, 4000]	12	[1650, 250]	17	[1650, 3750]
3	[825, 1000]	8	[206.25, 500]	13	[1650, 500]	18	[412.5, 1312.5]
4	[1650, 2000]	9	[412.5, 2000]	14	[1650, 1000]	19	[825, 2625]
5	[412.5, 250]	10	[412.5, 4000]	15	[412.5, 937.5]	20	[2475, 2625]
synthetic VM types							
1	[719, 2005]	4	[1135, 3542]	7	[1363, 2634]	10	[2100, 3013]
2	[917, 951]	5	[1231, 1989]	8	[1648, 1538]		
3	[1032, 1009]	6	[1311, 3238]	9	[2047, 1181]		

5.2 Benchmark Algorithms

FF&BF/FF [12, 23] uses three heuristics to allocate containers. It uses First Fit heuristics to allocate both containers and VMs and applies a Best Fit (BF) for selecting VM types. Whenever no available VM can host a container, the BF selects a type of VM which has just enough resource to host the container. Explicitly, BF selects the VM which has the minimum normalized free resources according to Eq.13.

$$Free\ resources = min\{\frac{\Omega^{cpu}(\tau_j) - \zeta^{cpu}(c_i) - \pi^{cpu}(\tau_j)}{\Omega^{cpu}(p_k)} \text{ and } \frac{\Omega^{mem}(\tau_j) - \zeta^{mem}(c_i) - \pi^{mem}(\tau_j)}{\Omega^{mem}(p_k)}\} \tag{13}$$

Dual-chromosome GA is a recent approach proposed in [20] to solve the resource allocation problem in container-based clouds. This approach uses a dual chromosome representation which includes two vectors, one represents a permutation of containers, the other represents the selected VM types. An individual requires a decoding process to construct the dual-chromosome into a solution. The rest of the algorithm follows a standard GA process with vector-based crossover and mutation operators.

This paper compares with two variations of the *dual-chromosome GA* with two decoding processes. The original work [20] applies a Next Fit (NF) decoding. We refer it as *DGA-NF* in the following content. We implement a different version that applies a First Fit (FF) decoding called *DGA-FF*.

In the experiments, we all also compare the wasted resources in the allocation. The wasted resources include all the free resources in both VMs and PMs as well as the overheads used by VMs (see Eq.14).

$$wasted\ resources = min\{\frac{\Omega^{cpu}(p_k) - \sum_{i=1}^{n} \zeta^{cpu}(c_i) \cdot x_{il}}{\Omega^{cpu}(p_k)} \text{ and } \frac{\Omega^{mem}(p_k) - \sum_{i=1}^{n} \zeta^{mem}(c_i) \cdot x_{il}}{\Omega^{mem}(p_k)}\} \tag{14}$$

5.3 Parameter Settings

The parameter setting of GGA and two dual-chromosome GAs are listed in Table 3. In addition to the operators that we proposed, we apply Elitism with size 5 and tournament selection with size 7. To ensure that all algorithms have the same computation time, we set the stopping criteria of all GAs to 12 seconds.

Table 3: Parameter Settings

Parameter	Description
runs	30
crossover	70%
mutation rate for dual-chromosome GA	10%
elitism	top 5 individuals
stopping criteria	12 seconds
Population	100
Selection	tournament selection (size = 7)

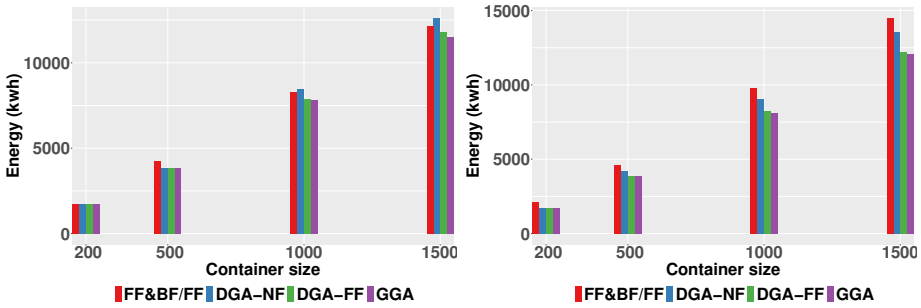
All algorithms were implemented in Java version 8 and the experiments were conducted on i7-4790 3.6 GHz with 8 GB of RAM running Linux Arch 4.14.15. We applied the Wilcoxon rank-sum to test the statistic significance.

5.4 Results

This section illustrates the performance comparison among the four algorithms in terms of energy consumption. Then, we explain the drawbacks of the compared algorithms by comparing the convergence, the number of VMs and the wasted resources in the allocation. Lastly, we compare the execution time of the four algorithms.

The energy consumption of four algorithms running for the same amount of time (12 seconds) are compared in Fig. 3 and Table.4. This ensures the comparison is fair. Our proposed *GGA* approach consistently achieves the best performance than the *FF&BF/FF* and two *dual-chromosome GA* approaches in large instances. The *DGA-FF* has a similar performance with *GGA* in the small instance (less than 1500 containers) but it performs poorly in the large instances. The *DGA-NF* performs better than *FF&BF/FF* in most of the instances except instance 3 and 4 (1000 and 1500 containers with synthetic VM types). In container 200 and container 500 instances, *DGA-FF* and *GGA* have similar performances. In larger instances, *GGA* has clearly show its advantages.

Due to the space limit, we show in Fig. 4 the convergence curves in terms of computation time from instance 4 and 8. In most instances except for instance 3 and 4, the convergence curves are similar to instance 8 where we may observe the *FF&BF/FF* is always flat because it has no searching process. *FF&BF/FF* is also easily affected by the VM types as it performs well in the synthetic VM types and performs poorly in the real-world data set. The *DGA-NF* starts with a much higher energy consumption than other algorithms. Although *DGA-NF*



(a) Synthetic VM types (b) Real-world VM types

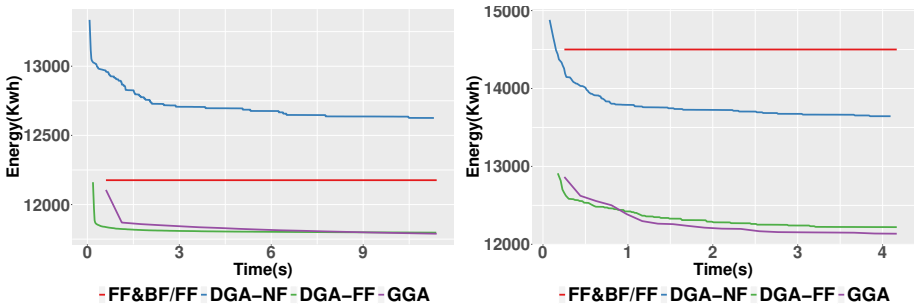
Fig. 3: Comparison of the average energy consumption

Table 4: Mean and standard deviation of the test instance with 95% confident interval.

	synthetic VM types			
	200	500	1000	1500
FF&BF/FF	1708.0 ± 0	4244.2 ± 0	8259.5 ± 0	12176.0 ± 0
DGA-NF	1685.6 ± 0.3	3838.5 ± 1.1	8,485.3 ± 94.1	12,625.8 ± 50.6
DGA-FF	1684.6 ± 0.2	3758.4 ± 151.1	7,865.7657 ± 1.0	11,795.3957 ± 1.5
GGA	1686.0 ± 0.1	3571.4 ± 177.1	7,833.9 ± 41.1	11,490.7 ± 108.8
	real-world VM types			
	200	500	1000	1500
FF&BF/FF	2093.2 ± 0	4635.0 ± 0	9809.2 ± 0	14500.4 ± 0
DGA-NF	1683.6 ± 0.4	4213.1 ± 1.8	9,027.9 ± 3.1	13,580.9 ± 93.9
DGA-FF	1682.3 ± 0.2	3827.8 ± 1.3	8,222.3681 ± 40.7	12,180.0944 ± 1.9
GGA	1683.1 ± 0.5	3828.2 ± 2.3	8,091.7 ± 91.1	12,083.8 ± 51.7

reaches convergence, its final fitness value cannot compete with the initial fitness from *DGA-FF* and *GGA*. In instance 3 and 4, the *DGA-NF* cannot outperform the *FF&BF/FF* approach. *DGA-FF* and *GGA* have a similar starting point. In instance 4, *DGA-FF* and *GGA* have a similar pattern while *GGA* outperforms in instance 8 after 1 second.

The major defect of *DGA-NF* is the decoding process. Compared to FF, NF closes a bin (such as VM and PM) whenever the current item (such as container and VM) cannot allocate to it while FF never closes a bin so that the future items can be still put into the unfilled bins. It means that NF cannot guarantee a VM is filled with containers. Consequently, we may observe *DGA-NF* starts from a bad allocation and takes a long time to converge. Replacing NF with FF immediately improves the performance. However, the *DGA-FF* is still inferior to the *GGA* approach.



(a) Instance 4 (synthetic VM types) (b) Instance 8 (real-world VM types)

Fig. 4: Comparison of the convergence

The number of VMs (left-hand side) and the wasted resources (right-hand side) are compared in Fig.5. The *FF&BF/FF* always uses the greatest number of VMs and has the highest wasted resources. For most instances, the *dual-chromosome* algorithms use fewer VMs and have fewer wasted resources except in instance 4. Our proposed *GGA* always uses the least number of VMs and has the least wasted resources.

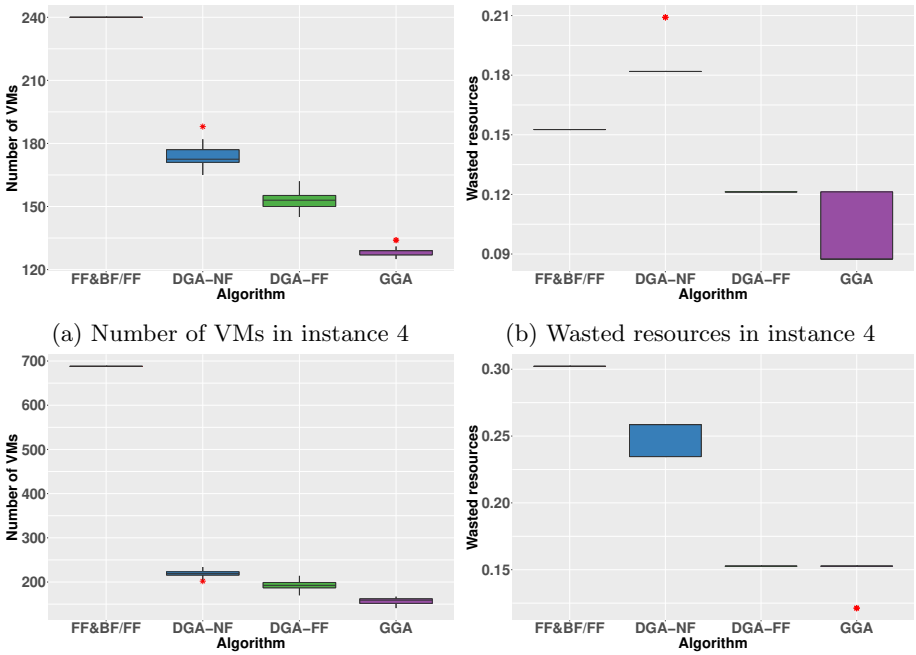
Due to the overheads and resource segmentation, the number of VMs generally proportional to the wasted resources. The *FF&BF/FF* always creates a VM that has the least resources to host a container, therefore, it creates a large number of small VMs. *DGA-NF* has a high wasted resource in instance 4 because *DGA-NF* cannot fill VMs with containers, hence, there are more free resources in VMs and PMs than the overheads. *DGA-FF* and *GGA* use fewer VMs. However, *DGA-FF* does not have the mechanism to reduce the number of VMs.

On the other hand, among all the algorithms *GGA*, can generate allocation solutions with the least wasted resources due to the merge operator. Without deliberately merging smaller VMs into larger ones, a PM could be filled with a large number of small VMs.

In summary, our propose *GGA* can find an allocation that leads to the least energy consumption in all the test instances. The performance of *dual-chromosome GA* varies with the decoding process.

6 Conclusion and Future Work

This work proposes a *Group GA (GGA)*-based approach to solve the resource allocation problem in container-based clouds. The experiments show that our proposed *GGA* approach outperforms three state-of-the-art approaches, a rule-based *FF&BF/FF* approach and two variations of *dual-chromosome GA* in terms of energy consumption. We propose three novel problem-specific operators, gene-level crossover, rearrangement, and unpack. These operators have shown effectiveness in searching good combinations of containers and VM types. Also, these



(a) Number of VMs in instance 4

(b) Wasted resources in instance 4

(c) Number of VMs in instance 8

(d) Wasted resources in instance 8

Fig. 5: Number of VMs and wastes in instance 4 and 8

operators can effectively search for better solutions directly on the representation. Current operators have a high computation cost in each generation, in the future, we will focus on improving the efficiency by applying clustering-based preprocessing approaches.

References

1. Bhandari, D., Murthy, C., Pal, S.K.: Genetic algorithm with elitist model and its convergence. International journal of pattern recognition and artificial intelligence **10**(06), 731–747 (1996)
2. Boxiong Tan, Hui Ma, Yi Mei: A NSGA-II-based approach for service resource allocation in Cloud. In: IEEE Congress on Evolutionary Computation (CEC). pp. 2574–2581 (2017)
3. Falkenauer, E.: A hybrid grouping genetic algorithm for bin packing. Journal of heuristics **2**(1), 5–30 (1996)
4. Garg, S.K., Buyya, R.: Green cloud computing and environmental sustainability. Harnessing Green IT: Principles and Practices **2012**, 315–340 (2012)
5. Guerrero, C., Lera, I., Juiz, C.: Genetic algorithm for multi-objective optimization of container allocation in cloud architecture. Journal of Grid Computing **16**(1), 113–135 (2018)

6. Kaaouache, M.A., Bouamama, S.: Solving bin packing problem with a hybrid genetic algorithm for VM placement in cloud. Procedia Computer Science **60**, 1061–1069 (2015)
7. Kaur, K., Dhand, T., Kumar, N., Zeadally, S.: Container-as-a-service at the edge: Trade-off between energy efficiency and service availability at fog nano data centers. IEEE Wireless Communications **24**(3), 48–56 (2017)
8. Koch, S., Wäscher, G.: A grouping genetic algorithm for the order batching problem in distribution warehouses. Journal of Business Economics **86**(1-2), 131–153 (2016)
9. Lin, M., Xi, J., Bai, W., Wu, J.: Ant colony algorithm for multi-objective optimization of container-based microservice scheduling in cloud. IEEE Access **7**, 83088–83100 (2019)
10. Liu, B., Li, P., Lin, W., Shu, N., Li, Y., Chang, V.: A new container scheduling algorithm based on multi-objective optimization. Soft Computing **22**(23), 7741–7752 (2018)
11. Liu, X.F., Zhan, Z.H., Deng, J.D., Li, Y., Gu, T., Zhang, J.: An energy efficient ant colony system for virtual machine placement in cloud computing. IEEE Transactions on Evolutionary Computation **22**(1), 113–128 (2016)
12. Mann, Z.Á.: Interplay of virtual machine selection and virtual machine placement. In: Lecture Notes in Computer Science. vol. 9846, pp. 137–151. Springer (2016)
13. Mann, Z.Á.: Resource optimization across the cloud stack. IEEE Transactions on Parallel and Distributed Systems **29**(1), 169–182 (2018)
14. Miller, B.L., Goldberg, D.E.: Genetic algorithms, tournament selection, and the effects of noise. Complex systems **9**(3), 193–212 (1995)
15. Piraghaj, S.F., Dastjerdi, A.V., Calheiros, R.N., Buyya, R.: A framework and algorithm for energy efficient container consolidation in cloud data centers. In: International Conference on Data Science and Data Intensive Systems. pp. 368–375. IEEE (2015)
16. Poon, P.W., Carter, J.N.: Genetic algorithm crossover operators for ordering applications. Computers & Operations Research **22**(1), 135–147 (1995)
17. Quiroz-Castellanos, M., Cruz-Reyes, L., Torres-Jimenez, J., Gómez, C., Huacuja, H.J.F., Alvim, A.C.: A grouping genetic algorithm with controlled gene transmission for the bin packing problem. Computers & Operations Research **55**, 52–64 (2015)
18. Şahin, M., Kellegöz, T.: An efficient grouping genetic algorithm for u-shaped assembly line balancing problems with maximizing production rate. Memetic Computing **9**(3), 213–229 (2017)
19. Shen, S., van Beek, V., Iosup, A.: Statistical characterization of business-critical workloads hosted in cloud datacenters. In: IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing. pp. 465–474. IEEE (2015)
20. Tan, B., Ma, H., Mei, Y.: Novel genetic algorithm with dual chromosome representation for resource allocation in container-based clouds. In: International Conference on Cloud Computing. pp. 452–456. IEEE (2019)
21. Wen, Y., Li, Z., Jin, S., Lin, C., Liu, Z.: Energy-efficient virtual resource dynamic integration method in cloud computing. IEEE Access **5**, 12214–12223 (2017)
22. Wolke, A., Bichler, M., Setzer, T.: Planning vs. dynamic control: Resource allocation in corporate clouds. IEEE Transactions on Cloud Computing **4**(3), 322–335 (2016)
23. Zhang, R., Zhong, A.m., Dong, B., Tian, F., Li, R.: Container-VM-PM architecture: A novel architecture for docker container placement. In: International Conference on Cloud Computing. pp. 128–140. Springer (2018)