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

Boxiong Tan, Hui Ma, and Yi Mei

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

Abstract. Containers have gained popularity because they support fast development and deployment of cloud-native software such as microservices and serverless 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 representations. To address these issues, we propose a novel group genetic algorithm (GGA) with a direct representation and problem-specific operators. This design has shown significantly better performance than the state-of-the-art algorithms in a wide range of test datasets.

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

1 Introduction

Container-based clouds [14] 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 apply server consolidation [21] 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 to 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.
Boxiong Tan, Hui Ma, and Yi Mei

which are NP-hard [22]. 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 [6, 11, 14, 22], and meta-heuristic algorithms [2, 4, 19] 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 the problem of allocating containers directly to PMs or uses indirect representation which is inefficient in the searching process.

Group Genetic Algorithm (GGA) was proposed by Falkenauer [3] and has inspired many studies in solving the VM allocation problem [10,20]. 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, GGA [3,16] can only solve one-level problems.

This research aims at proposing a novel GGA for the RAC problem to minimize the energy consumption. The proposed GGA 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. 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 [22] and two variations of dual-chromosome GAs [19].

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. 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 approaches, rule-based approaches, and meta-heuristics approaches. Piraghaj [14], Kaur [6], Mann [12], Liu [9] and Zhang [22] 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. [21] suggest that 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 either focus on one-level allocation problem such as [4, 8], or uses an indirect representation [2, 19]. Guerrero et al. [4] propose an NSGA-II-based approach for a four-objective allocation problem. Lin et al. [8] 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, 19] propose two meta-heuristic approaches for the RAC problem, an NSGA-II-based and a dual-chromosome GA (DGA). 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 [4, 8] only consider the one-level structure which inherently leads to local optimal solutions. The second drawback is that the decoding process of [19] 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 [15]. 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 [16] 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 [7], VM placement problem [5, 10], and
assembly line balancing problem [17]. However, it has not been 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 ($RAC$) is a task of allocating a set of containers to a set of VMs of 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 $\tau_j$ 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:

$$E = \sum_{k=1}^{K} E_k,$$  \hspace{1cm} (1)

where $E_k$ is the energy consumption of the $k$th PM ($K$ is the number of PM used).

$E_k$ is calculated as follows:

$$E_k = E_k^{idle} + (E_k^{full} - E_k^{idle}) \cdot \mu_{cpu}^k,$$ \hspace{1cm} (2)
where $E_{idle}^k$ and $E_{full}^k$ indicate the energy consumption of the $k$th PM per time unit if it is idle and fully loaded, respectively. $\mu_{cpu}^k$ indicates the CPU utilization level of the $k$th PM. $\mu_{cpu}^k$ is calculated as follows.

$$\mu_{cpu}^k = \frac{\sum_{l=1}^{L} \left( \sum_{j=1}^{m} \pi_{cpu}(\tau_j) \cdot z_{jl} + \sum_{i=1}^{n} \Omega_{cpu}(c_i) \cdot x_{il} \right) \cdot y_{lk}}{\Omega_{cpu}(p_k)},$$

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 $l$th created VM, and 0 otherwise. $y_{lk}$ takes 1 if the $l$th created VM is allocated to the $k$th PM, and 0 otherwise. $z_{jl}$ takes 1 if the $l$th 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.

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

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

$\sum_{k=1}^{K} y_{lk} = 1, \forall l = 1, \ldots, L$, \hspace{1cm} (6)

$\sum_{j=1}^{m} z_{jl} = 1, \forall l = 1, \ldots, L$, \hspace{1cm} (7)

$\sum_{i=1}^{n} \zeta_{res}(c_i) x_{il} \leq \sum_{j=1}^{m} \Omega_{res}(\tau_j) z_{jl}$, \hspace{1cm} (8)

$\forall l = 1, \ldots, L$, \hspace{0.5cm} res $\in \{cpu, mem\}$,

$\sum_{l=1}^{L} \sum_{j=1}^{m} \Omega_{res}(\tau_j) z_{jl} \leq \Omega_{res}(p_k)$, \hspace{1cm} (9)

$\forall k = 1, \ldots, K$, \hspace{0.5cm} res $\in \{cpu, mem\}$,

$x_{il}, y_{lk}, z_{jl} \in \{0, 1\}$, \hspace{1cm} (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 (see Eq.1) will be used as 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

<table>
<thead>
<tr>
<th>Input</th>
<th>a set of containers, a set of VM types, a list of PMs,</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output</td>
<td>an allocation of containers</td>
</tr>
<tr>
<td></td>
<td>population ← Initialization;</td>
</tr>
<tr>
<td>gen</td>
<td>for gen does not reach the maximum generation do</td>
</tr>
<tr>
<td></td>
<td>fitness evaluation(population);</td>
</tr>
<tr>
<td></td>
<td>new population ← elitism(population);</td>
</tr>
<tr>
<td></td>
<td>while has not fill the new population do</td>
</tr>
<tr>
<td></td>
<td>parents ← tournament selection(population);</td>
</tr>
<tr>
<td></td>
<td>children ← gene-level crossover(parents);</td>
</tr>
<tr>
<td></td>
<td>unpack(children);</td>
</tr>
<tr>
<td></td>
<td>merge(children);</td>
</tr>
<tr>
<td></td>
<td>add children to the new population</td>
</tr>
<tr>
<td>gen</td>
<td>gen ← gen + 1;</td>
</tr>
<tr>
<td></td>
<td>end</td>
</tr>
<tr>
<td></td>
<td>return an allocation of containers;</td>
</tr>
</tbody>
</table>

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 evolutions where each loop is called a generation. In each generation, individuals are evaluated with a fitness function (Eq.(1)). For this algorithm, the top individuals are preserved and copied to the new population with Elitism [1]. Tournament selection [13] is used to direct the population to the high-fitness region. Then, we propose 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. Details of the three operators will be presented later.

4.2 Representation

We use an individual (see Fig. 1) to represent a complete solution for a RAC problem. A 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 [19] 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 the First-Fit heuristic. 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 that 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 [16]. Similarly, highly utilized PMs could lead to fewer PMs in the allocation problem. Therefore, good gene 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 pairwise by utilization (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.

An example of the gene-level crossover is shown in Fig. 2. 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 heuristic uses FF to allocate containers to existing VMs. If no VM is available, we randomly create a new VM and allocate containers to it. Then, we use FF to allocate the new VM to PMs.

\[
R = \frac{\zeta_{cpu}(c_i)}{\Omega_{cpu}(p_k)} \cdot \frac{\zeta_{mem}(c_i)}{\Omega_{mem}(p_k)}
\]

Our rearrangement operator is inspired by [16] to avoid the drawback of First-Fit (FF) and further improve the allocation of a VM. In the bin packing problem, FF-based approaches [3,16] have been widely used. However, a simple FF-based approach cannot change the existing packing of a bin. Hence, a replacement heuristic is developed. The core idea of the replacement heuristic is that the smaller items are easier to allocate. Therefore, we can replace a big container with smaller ones, which can be easily allocated to existing VMs without creating a new VM.

4.6 Unpack

Unpack operator eliminates low-utilized PMs and reallocates their containers. This operator prevents premature convergence and introduces new gene components into the current population.
Algorithm 2: Rearrangement operator

Input: a target container, a list of PMs,
Output: a list of PMs

1. Sort the containers in all VMs according to Eq.11 in ascending order;
2. for each VM do
   3. if the two smallest containers in each VM can be replaced by the target container then
      4. Replace two containers with the target VM;
      5. Allocate two containers using FF&RC/FF;
      6. return a list of PMs;
   end
8. end
9. Allocate the target container using FF&RC/FF;
10. return a list of PMs;

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

\[
probability = \frac{1 - \Omega_{cpu}(p_k)}{\sum_{k=1}^{K} 1 - \Omega_{cpu}(p_k)}
\]  (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 through some generation, highest 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 here refer 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 for which a 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 contain an increasing number of containers (from 200 to 1500) and two sets of VM types. We use a real-world application trace (AuverGrid trace [18]) 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 a set of resource requirements and use them to define the containers to be allocated.

For the settings of PMs and VMs, we assume homogeneous PMs which have 8 cores and a total capacity of [13200 MHz, 16000 MB]. The maximum energy consumption for the PM is set to 540 KWh the same setting as [11]. 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 whereas the synthetic ones are random. The values of 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

<table>
<thead>
<tr>
<th>Instance</th>
<th>VM types</th>
<th>Number of containers</th>
<th>Instance</th>
<th>VM types</th>
<th>Number of containers</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>synthetic VM types</td>
<td>200</td>
<td>5</td>
<td>real-world VM types</td>
<td>200</td>
</tr>
<tr>
<td>2</td>
<td>synthetic VM types</td>
<td>500</td>
<td>6</td>
<td>real-world VM types</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>synthetic VM types</td>
<td>1000</td>
<td>7</td>
<td>real-world VM types</td>
<td>1000</td>
</tr>
<tr>
<td>4</td>
<td>synthetic VM types</td>
<td>1500</td>
<td>8</td>
<td>real-world VM types</td>
<td>1500</td>
</tr>
</tbody>
</table>

5.2 Benchmark Algorithms

FF&BF/FF [11,22] 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 existing VM can host a given 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.

$$Free\ resources = \min \left\{ \frac{\Omega^{cpu}(p_k) - \zeta^{cpu}(c_i) - \pi^{cpu}(c_j)}{\Omega^{cpu}(p_k)} \quad \text{and} \quad \frac{\Omega^{mem}(p_k) - \zeta^{mem}(c_i) - \pi^{mem}(c_j)}{\Omega^{mem}(p_k)} \right\}$$  

Eq.13
Dual-chromosome GA is a recent approach proposed in [19] 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 [19] 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 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).

\[
\text{wasted resources} = \min \left\{ \frac{\Omega^{cpu}(p_k) - \sum_{i=1}^{n} \zeta^{cpu}(c_i) \cdot x_{il}}{\Omega^{cpu}(p_k)} \text{ and } \frac{\Omega^{mem}(p_k) - \sum_{i=1}^{n} \zeta^{mem}(c_i) \cdot x_{il}}{\Omega^{mem}(p_k)} \right\}
\] (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 (all algorithms finished in this period of time).

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
Table 3: Parameter Settings

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>runs</td>
<td>30</td>
</tr>
<tr>
<td>crossover</td>
<td>70%</td>
</tr>
<tr>
<td>mutation rate for dual-chromosome GA</td>
<td>10%</td>
</tr>
<tr>
<td>elitism</td>
<td>top 5 individuals</td>
</tr>
<tr>
<td>stopping criteria</td>
<td>12 seconds</td>
</tr>
<tr>
<td>Population</td>
<td>100</td>
</tr>
<tr>
<td>Selection</td>
<td>tournament selection (size = 7)</td>
</tr>
</tbody>
</table>

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, DGA-NF and DGA-FF, 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 instances with 200 and 500 containers, DGA-FF and GGA have similar performances. In larger instances, GGA has clearly show its advantages.

![Figure 3: Comparison of the average energy consumption](image)

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 instance 3 and 4, the convergence curves are similar to instance 8 where we observe the FF&BF/FF shows a flat line because it has no searching process. FF&BF/FF is also easily affected by the set of available VM types as it performs well in
Table 4: Mean and standard deviation of the test instances with 95% confident interval.

<table>
<thead>
<tr>
<th></th>
<th>synthetic VM types</th>
<th></th>
<th>real-world VM types</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>200</td>
<td>500</td>
<td>1000</td>
<td>1500</td>
</tr>
<tr>
<td>FF&amp;BF/FF</td>
<td>1708.0 ± 0</td>
<td>4244.2 ± 0</td>
<td>8259.5 ± 0</td>
<td>12176.0 ± 0</td>
</tr>
<tr>
<td>DGA-NF</td>
<td>1685.6 ± 0.3</td>
<td>3838.5 ± 1.1</td>
<td>8485.3 ± 94.1</td>
<td>12625.8 ± 50.6</td>
</tr>
<tr>
<td>DGA-FF</td>
<td>1684.6 ± 0.2</td>
<td>3758.4 ± 151.1</td>
<td>7865.7657 ± 1.0</td>
<td>11795.3957 ± 1.5</td>
</tr>
<tr>
<td>GGA</td>
<td>1686.0 ± 0.1</td>
<td>3571.4 ± 177.1</td>
<td>7833.9 ± 41.1</td>
<td>11490.7 ± 108.8</td>
</tr>
</tbody>
</table>

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. Even though replacing NF with FF can improve the performance of DGA. However, the DGA-FF is still inferior to the GGA approach.

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 is generally proportional to the wasted resources. The FF&BF/FF always creates a VM that has the least resources to host a container, and therefore, creates a large number of small VMs. DGA-NF has a high wasted resource in instance 4 because DGA-NF cannot fill VMs with container, hence, there are more free resources in VMs and PMs than the overheads of VMs. 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 thanks 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 [19] in terms of energy consumption. For our GGA, 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 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

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

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Number of VMs</th>
<th>Wasted resources</th>
</tr>
</thead>
<tbody>
<tr>
<td>FF&amp;BF</td>
<td>120</td>
<td>0.09</td>
</tr>
<tr>
<td>DGA-NF</td>
<td>150</td>
<td>0.12</td>
</tr>
<tr>
<td>DGA-FF</td>
<td>180</td>
<td>0.15</td>
</tr>
<tr>
<td>GGA</td>
<td>210</td>
<td>0.18</td>
</tr>
<tr>
<td>FF&amp;BF</td>
<td>240</td>
<td>0.21</td>
</tr>
</tbody>
</table>