# Kapellmeister – Multi-Objective Optimization of Container Allocation and Rescheduling with Genetic Algorithm

Artemii Velikanov 20160884 Kunduz Baryktabasova 20170752 Shynar Torekhan 20160882 Timothy M. Mathews 20170879

Optimizing container resource allocation is of critical importance due to the emergence of containerized systems. Here, we propose a tool based on a genetic algorithm for container resource allocation and optimization. We provide an extendable tool with scheduling and rescheduling tasks, which could be easily adjusted to fit company's optimization objectives of interest. **Repository of the project** is available here: `https://github.com/Shynar88/Kapellmeister-Multi-Objective-Optimization-of-Container-Allocation-and-Rescheduling-with-GA`

## I. Introduction

According to recent Gartner report, "By 2023, more than 70% of global organizations will be running more than two containerized applications in production..." Thus, with the spread of microservice architecture, IoT and big data which take advantage of encapsulating system's components into containers, the importance of container management tools arises. Container resource allocation influences system performance and resource consumption, and thus is a key factor for cloud providers. Resource management optimization is an NP-complete problem(optimal solution can be only obtained through an evaluation of all possible combinations), and, thus, must be addressed by meta-heuristic approach.

The precise goals of our tool is Scheduling and Rescheduling tasks. Where 'Scheduling' task means - given a set of containers and a set of nodes, find optimized allocation which satisfies objectives, and 'Rescheduling' stands for a task where given an allocation of containers and nodes, and a new container which does not fit in, rearrange the allocation to fit the new container without allocating additional node. Node stands for machine on which the container is placed.

This research area is relatively recent. There are only 2 papers which use Genetic Algorithm for Container Scheduling. Guerrero et al.(6 November 2017)[2] - was the first to use GA in container scheduling. Then, Imdoukh et al.(23 October 2019)[3] - enhanced results of Guerrero by proposing different chromosome structure and using NSGA-III instead of NSGA-II.

The contributions of our project are fivefold. **First,** we are the first to apply Search-Based technique on Rescheduling problem. **Second,** released the first public implementation of Genetic Algorithm applied on Scheduling problem, written from scratch, which beats the performance of Kubernetes by 71%. **Third,** modified NSGA-III for container allocation task. **Fourth,** released various visualization and logging scripts for understanding the progress of Genetic Algorithm. **Fifth,** provided well-documented open source tool which can be easily extended to include optimization objectives suitable for user's needs and requirements.

## II. Methodology

### A. Chromosome Representation

We used Chromosome Representation from Imdoukh et al.[3] Chromosome is represented by the list of containers and the list of nodes, which indicates to which node is container assigned. Container could be unassigned.

| Node ID | $n_1$ | $n_2$ | $n_3$ | $n_1$ | $n_2$ | $n_1$ | $n_2$ | | $n_4$ |
|---|---|---|---|---|---|---|---|---|---|
| Container ID | $c_1$ | $c_2$ | $c_3$ | $c_4$ | $c_5$ | $c_6$ | $c_7$ | $c_8$ | $c_9$ |

FIG. 1. Chromosome Representation

### B. NSGA-III

NSGA-III (Nondominated Sorting Genetic Algorithm III) is a further development of NSGA-II. It belongs to a class of Evolutionary Multiobjective Optimization algorithms.

It is used to find a set pareto optimal solutions in a multidimensional objective space. NSGA-II was limited to 2-3 objectives in practice, which is addressed by a new diversity-preservation mechanism of selection in the updated method. To further read on NSGA-III algorithm refer to [4] .

In default NSGA-III one solution dominates the other if all of its objectives are smaller. For our purposes we use modified domination function. If A is more feasible than B, A dominates B. If they both have same feasibility, they are compared by their objective values [3].

### C. Selection

For the selection the Tournament selection is used. The winner is the most feasible solution. Where Feasibility score is represented number of nodes violating CPU and Memory constraints. Thus, the lower the Feasibility score, the better. In case of tie, the winner is selected randomly among solutions with the least feasibility score.

### D. Crossover

For the crossover - single point crossover is used as described in Imdoukh et al.[3] The crossover is done by selecting the crossover point in the nodes list of parent Chromosomes. Genes to the sides of crossover point are swapped between the two parent chromosomes as shown in Figure 2.

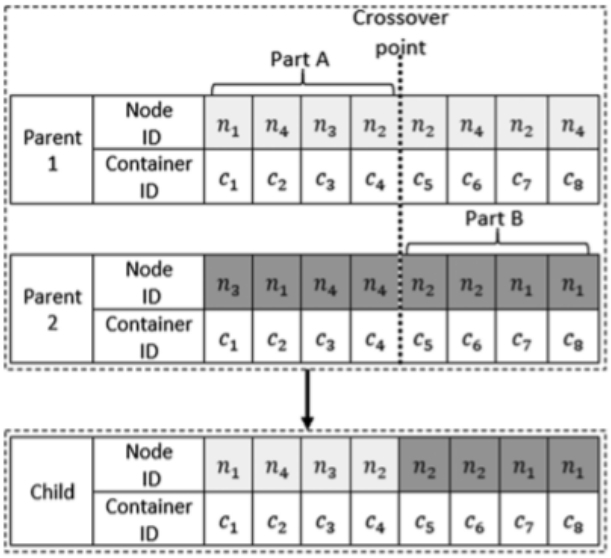


FIG. 2. Single point crossover

### E. Mutation

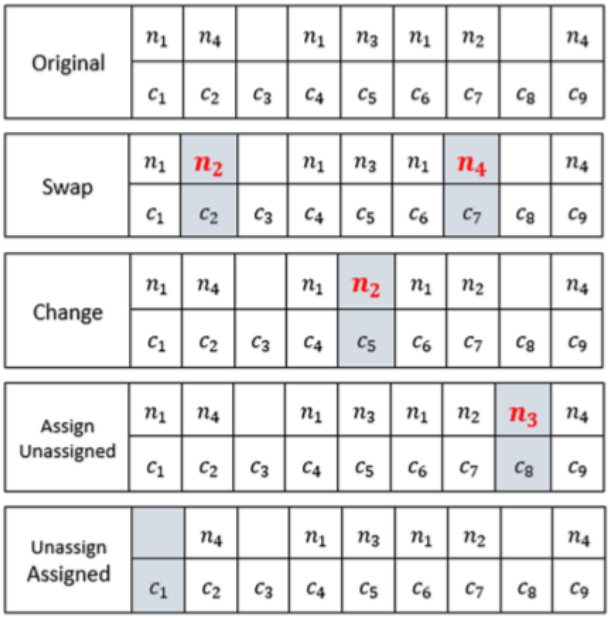


FIG. 3. Four types of mutations

The four types of mutations described in Imdouch et al.[3] were implemented. They are presented in Figure 3. Swap mutation swaps two nodes between containers. Change mutation changes the assigned node to any other. Assign Unassigned mutation assigns unassigned container to random node. And Unassign Assigned mutation unassigns node from container.

### F. Objective Functions

In any GA algorithm evolution of an individual chromosome is needed. Fitness Function evaluates how close a given solution is to the optimum solution of the desired problem. It determines how fit a solution is. In this work we focus on solutions where containers are equally distributed distributed across the nodes, containers with same service are equally distributed distributed, power efficient, utilize most of available resources, and assign maximum number of tasks. Therefore, five objective functions are defined to evaluate the solutions where objective values must be minimized to achieve optimal individual criteria. This five objective functions were inspired from Mahmoud Imdoukh's paper[3]. For rescheduling task we added one more objective function that evaluates how chromosome was changed compare with ancestor.

#### 1. *Equal task distribution objective function*

Equal distribution objective function evaluates how well tasks are divided across the nodes. Containers should be evenly distributed across nodes so that in a case of failure, minimum number of containers are lost. Let's consider node which has 4 containers on it, then this node will be assigned to 1+2+3+4=10(triangular function). So for a given cluster of nodes fitness function value is summation of each node's value. The algorithm ignores the idea of task's replicas and considers all tasks to be different. Below you could see the pseudocode.

```
def off_1(self):
  v = 0
  for node in nodes:
    t = 0
    i = 0
    for each in node.containers_list:
      i+=1
      t+=i
    v += t
  return v
```

For better understanding this objective, let's consider figure 4. It shows two different clusters of nodes with same amount of nodes and containers, however distribution of containers are different. The left cluster is an example of unequal distribution with objective value F1(a)= (1+2+3+4) + (0) + (1+2) + (1) = 14, the right F1(b)= (1+2) + (1) + (1+2) + (1+2) = 10 is equally distributed.

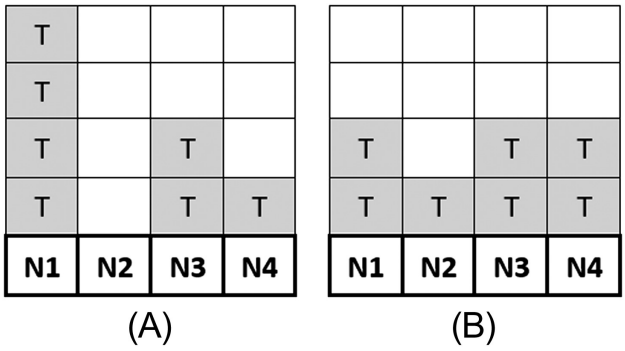


FIG. 4. Equal distribution example

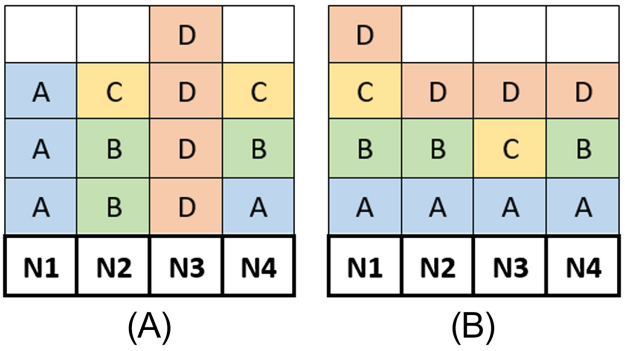


FIG. 5. Unique distribution example

### 2. *Unique distribution objective function*

Unique distribution objective function is used to ensure that each node is assigned a different set of containers. So, that replicas of container are spread among the nodes. Failure of a node should not lead to complete system failure. Each node is evaluated as a sum of triangular functions for each container. Below you could see the pseudocode.

```
def off_2(self):
  v = 0
  for node in nodes:
    dic = {}
    for container in node.containers_list:
      if(dic.get(container.task_type) \
            == None):
         dic[container.task_type] = 1
      else:
         dic[container.task_type] += 1
    for key in dic:
       n = dic[key]
       v += (n+1)*n/2
  return v
```

For better understanding this objective, let's consider figure 5. It shows two different clusters of nodes with same amount of nodes, containers and each container has same replicas, however distribution of replicas are different. On the left cluster same replicas mostly allocated at one node. It's objective value is F2(a)= (6)+(3+1)+(10)+(1+1+1+1)=23. On the right cluster replicas are evenly distributed among nodes. So, in case if any node failure we still have replicas of the containers on other nodes. The objective value of the right cluster is F2(b) = (1+1+1+1) + (1+1+1) + (1+1+1) + (1+1+1) = 13. Thus solution on the right is better.

### 3. *Power consumption objective function*

The power consumption of each cluster of nodes is estimated using the linear power model, as shown in below pseudocode. Power estimation composed of three main factors. The first one is the average power consumed by node n when its resources are fully utilized and no more tasks can be assigned to it. The second factor is the average power consumed by node n when it is not assigned to any task, idle state. The third one is the average of CPU and memory percentage usages. The lower the objective value of cluster, the more power efficient the schedule is.

```
def off_3(self):
  v = 0
  for n in nodes:
    c = (n.cpu_spec- n.rem_cpu)/ \
          n.cpu_spec
    m = (n.mem_spec - \
          n.rem_memy)/n.memy_spec
    p = (n.max_pow - n.idle_pow) \
        * (c + m)/2 + n.idle_pow
    v += p
  return v
```

In the figure 6, the the right cluster has F3(b)= 132.5 + 135.5 + 132 + 134.75 = 534.75 and left cluster has F3(a) = 119 + 121.5 + 144 + 159.5 = 544 . Thus right solution is better, since load was transferred from nodes with bigger power consumption to nodes with less consumption.

| Max Power | 140 | 160 | 180 | 200 |
|---|---|---|---|---|
| Idle Power | 80 | 90 | 100 | 110 |
| Memory Usage | 80% | 25% | 75% | 95% |
| CPU Usage | 50% | 65% | 35% | 15% |
| Node ID | N1 | N2 | N3 | N4 |

(A)

| Max Power | 140 | 160 | 180 | 200 |
|---|---|---|---|---|
| Idle Power | 80 | 90 | 100 | 110 |
| Memory Usage | 85% | 95% | 55% | 40% |
| CPU Usage | 90% | 35% | 25% | 15% |
| Node ID | N1 | N2 | N3 | N4 |

(B)

FIG. 6. Power consumption example

*4. Resources utilization balancing objective function*

This objective function evoluates efficiency of resourse usage. So that it helps to select cluster where each nodes have same proportion of memory and cpu usage. So it eliminates solutions where cpu if fully utilies while memory not and help evenly distribute the load. Below pseudocode proposed by proposed by Gao[6].

```
def off_4(self):
  v = 0
  i = 0
  node_ids = self.node_ids
  for node_id in node_ids:
    if (node_id == None):
      continue
    n = nodes[node_id]
    i += 1
    v += abs(n.rem_cpu/n.cpu_spec - \
            n.rem_mem/n.mem_spec)
  if (i == 0):
    return 0
  return 100*v/i
```

At the figure 7, after calculation of objective values for each cases F4(a) = 31.25 and F5(b) = 16.25, we could notice that right solution is since load is evenly spread and proportion of memory and cpu usage is almost same.

| Memory Usage | 80% | 25% | 75% | 95% |
|---|---|---|---|---|
| CPU Usage | 50% | 75% | 45% | 40% |
| Node ID | N1 | N2 | N3 | N4 |

(A)

| Memory Usage | 65% | 75% | 45% | 90% |
|---|---|---|---|---|
| CPU Usage | 55% | 60% | 35% | 60% |
| Node ID | N1 | N2 | N3 | N4 |

(B)

FIG. 7. Resources utilization balance example

*5. Unassigned tasks reduction objective function*

The fifth objective function, evaluates the number of unassigned tasks in a given solution. This objective helps to remain number of unsigned containers small. Since UnassignAssigned mutation tends to increase number of unassigned containers while minimizing above four fitness objectives. The solution will represent maximum number of containers that could be assigned to cluster's nodes while optimizing the remaining objectives.

```
def off_5(self):
  node_ids = self.node_ids
  v = 0
  for node_id in node_ids:
    if (node_id == None):
      v += 1
      continue
  return v
```

In Figure 8, right solution is better than left solution as more tasks are assigned to available nodes and thus reducing the number of unassigned tasks.

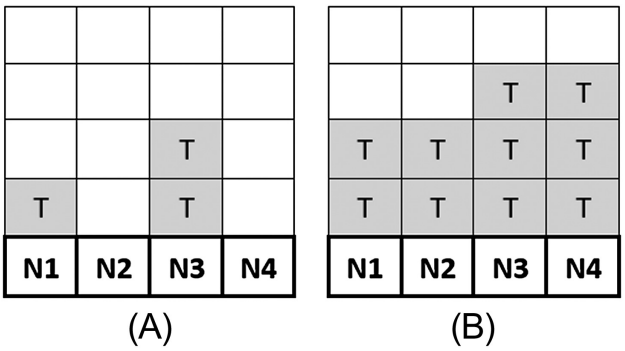


FIG. 8. Unassigned task reduction example

*6. Containers permutation reduction objective function*

While above objective functions were inspired from Mahmoud Imdoukh work [3], this objective was designed by us, since no Rescheduling using GA was done before. This objective shows how differ two clusters, in sense of containers allocation. The motivation behind this objective function was that movement of containers includes the undesirable overhead and fitting the new node at the cost of moving the majority of containers is undesired. Rescheduling is done when adding new container there is no node where it might be assigned, however total remaining CPU and memory allows to allocate new container. In that sense our system should reshuffle given allocation so that new configuration will be valid. For evaluation of different permutation we choose the one which did less movement, since less work should be done to support most similar scheduling. Implementation of this objective is bit different, since we pass chromosome before rescheduling(the existing allocation where new container cannot be allocated) and compare it with computed chromosomes.

```
def off_6(self, init_chromosome):
  node_ids = self.node_ids
  init_node_ids = init_chromosome.node_ids
  i = 0
  v = 0
  for j in range (len(node_ids)-1):
    if (node_ids[j] != init_node_ids[i]):
      i +=1
      continue
    v += 1
    i += 1
  return v
```

Figures 9, 10, 11 illustrates rescheduling of cluster A. We got two examples of rescheduling B and C. Since clus-

ter B has less container movements than C, it is considered better solution. The objective value of cluster B F6(b)= 1 and for C F6(c) = 6. Thus, the less the objective value the better a solution.

| | | | |
|---|---|---|---|
| C | C | D | D |
| B | B | C | C |
| A | A | A | A |
| N1 | N2 | N3 | N4 |

FIG. 9. Containers permutation reduction example

| | | | |
|---|---|---|---|
| | | | B |
| C | C | D | D |
| B | F | C | C |
| A | A | A | A |
| N1 | N2 | N3 | N4 |

FIG. 10. (B) permutation example

| | | | |
|---|---|---|---|
| F | | | |
| C | D | D | C |
| A | C | B | C |
| A | A | A | B |
| N1 | N2 | N3 | N4 |

FIG. 11. (C) permutation example

## III. Results and Evaluation

### A. Experiment Overview

There are 6 key steps involved in running this experiment

1. Creating a set of nodes (with specifications of physical machines)
2. Creating a set of containers (with specifications of services)
3. Generating an allocation in accordance with Kubernetes allocation policies
4. Performing GA and selecting a single allocation
5. Comparing the fitness values of the allocation from Kubernetes and the chosen allocation from GA
6. Comparing the fitness values of the rescheduling allocation from scheduling allocation with additional container.

### B. Creation of nodes

We generate nodes that follow the specifications of A1, M4 and M5a instance types from AWS. Instance types consist of a set of instances that each have different number of cores and amount of memory. We use one instance type for each experiment and generate an equal number of nodes with the specifications of each kind of instance in an instance type.

### C. Creation of containers

We create containers with CPU and memory requirements that are randomly generated within certain upper and lower bounds, as in Imdoukh et al. We have different upper and lower bounds for the experiment. Up to 3 replicas of each task are created to simulate how a service can have multiple instances.

### D. Selecting an allocation from the optimal front

The allocation with the lowest average normalised objective value is selected from the optimal pareto front after performing NSGA-3. Normalisation is done by using a slightly modified version of formula 15 from Guerrero et al.
$\text{SOV} = \sum_{i=1}^{n} \frac{1}{n} * \frac{f_i(allocation) - min_i}{max_i - min_i}$
where i is the current objective, n is the total number of objectives, $min_i$ and $max_i$ are the lowest and highest recorded value of that objective amongst all solutions in the front and $f_i(allocation)$ is the fitness of the current allocation in the $i^(th)$ objective SOV stands for single objective value. The SOV of every solution in the front is calculated and the solution with the smallest SOV is selected.

### E. Comparison between Kubernetes and GA

Once we have the set of nodes and the set of containers, using the two Kubernetes policies of PodFitResources and LeastRequestPriority highlighted in Guerrero et. al we can get the allocation of containers to nodes that Kubernetes would make.
Next, we pass the exact same sets of nodes and containers for performing GA. Non-dominated sort is performed on the final population. The optimal front after sorting is then collected. The selection method described in section D is used to pick an allocation from the front as the representative solution from GA.

The fitness values on each of the 5 objectives is computed for both the kubernetes allocation the chosen GA allocation. The results are visualised as a bar graph comparing the values on each objective between both allocations and saved to the directory.

### F. Comparison rescheduling with scheduling with additional container

The Kubernetes does not provide rescheduling for containers allocation. That is why we choose different approach for evaluating rescheduling algorithm. As described before we created nodes and containers and achieved best allocation using simple NSGA-III algorithms using 5 objective functions. After that we added new container so that there is no node which will fit it, and rescheduling must be done. From this point, firstly, we run Rescheduling algorithm and pass as a parameter the allocation(obtained from run of NSGA explained above) which cannot fit in the new container. Secondly, we run simple Scheduling algorithm, as it was done in step 4 of the evaluation. In other words, we run scheduling on empty nodes, so that it returns allocation which includes the new container which previously didn't fit in. Finally, two computed allocations - one from Scheduling and another from Rescheduling were compared by new 6th objective function which was introduced before - Containers permutation reduction objective function.

### G. Results

#### 1. *Scheduling Results*

On the following plots you can see comparison between our algorithm results and default Kubernetes allocation algorithm. Keep in mind that every objective is meant to be minimized so lower is always better. Since our result is a Pareto front, we can choose different priorities for our allocation. First, we allocate all containers to nodes, we get Figure 12. As mentioned earlier, all objectives are for minimisation, so lower is better. We were able to get a 9% decrease on the first objective and a 5% decrease on the third. This came at the cost of a 80% increase in the third and a negligible increase in the fifth objective.

We can also prioritise other objectives sacrificing full allocation as demonstrated on Figure 13. In this case, we were able to get a 53% decrease on the first objective, 42% decrease on the second and a 5% decrease on the third. This came at the cost of a negligible increase in the fourth and 78 unassigned tasks

Figure 14 and 16 show the dynamics of training process. Lower bound decreases for most of the tasks, which means that some solutions have improved fitness values on these tasks. This is the main criterion to evaluate the success of the algorithm. Increasing upper bound does not bother us too much because it just shows that the search space is being explored. Blue average line is there to show where the density of the population tends to,

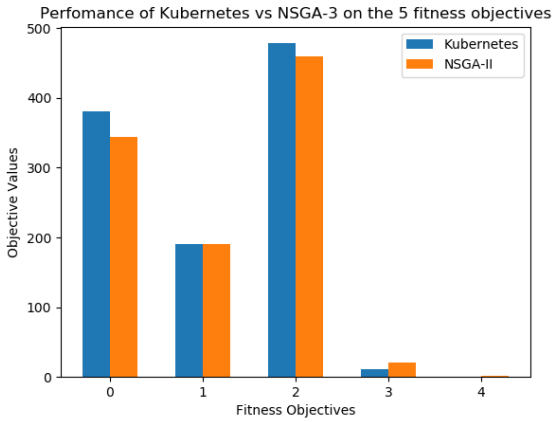


FIG. 12. Kubernetes and NSGA results compared, result with minimized unallocated containers is used

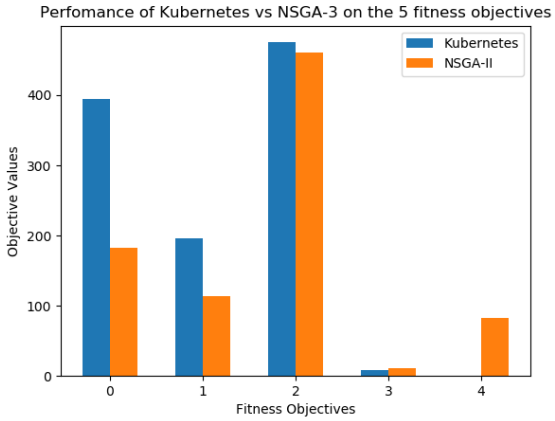


FIG. 13. Kubernetes and NSGA results compared, result with smaller objectives 1 and 2 used

upper or lower bound.

#### 2. *Rescheduling Results*

Results for rescheduling are quite promising. On the sixth objective, the rescheduled allocation has 71% lower value. In other words, rescheduling is able to minimise the movement of tasks around the nodes by 71% compared to scheduling from the beginning. As the trade-off, the rescheduled allocation is 65% higher on the first objective value, 34% higher on the second objective. Additionally there is a beneficial 57% decrease in the fifth objective. As shown in graph Fig 15, objective function for containers permutation(movement) reduction in Rescheduling algorithm shows better results than simple scheduling with inserted container. Which means that Rescheduling algorithm keeps cluster as much as conserved when new container is added(minimizes the con-

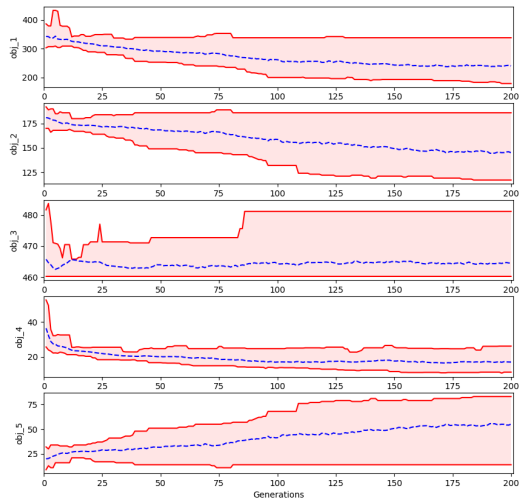


FIG. 14. Objective values during training process (red field bounds objective values of population, blue line is their average value)

tainer movement).

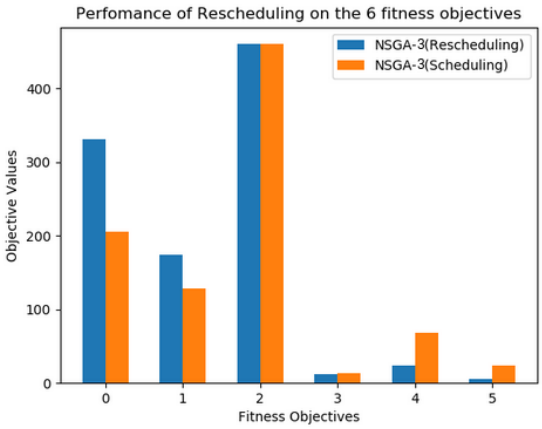


FIG. 15. Scheduling and rescheduling results compared

## IV. Challenges

- Wrote Genetic Algorithm for Scheduling task following the abstract description in Imdoukh et al.[3]
- Needed to make it possible to compare Kubernetes and GA. Their representations needed to be made compatible. A Kubernetes allocation needed to be treated as a chromosome as well.
- Had to invent, design and develop optimization objective from scratch for Rescheduling task.

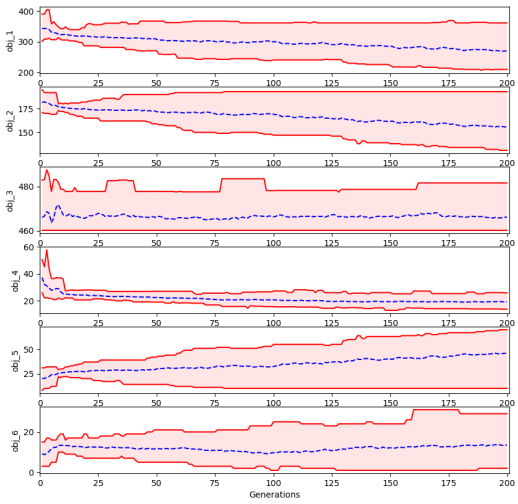


FIG. 16. Training process for 6 objectives

## V. Roles and Responsibilities

### A. Artemii Velikanov

Implemented interface between NSGA-III library and our custom problem-specific objects. Generally debugged the code.

Github ID: Electr0phile

### B. Shynar Torekhan

Implemented mutations, crossover, creating initial population, selection(for mating pool). Wrote skeleton code and initial implementation of classes. Wrote logging feature. Extended the scheduling code to include the rescheduling task(modified class interfaces and genetic operations to support rescheduling).

Github ID: Shynar88

### C. Timothy M. Mathews

Wrote the code to run and evaluate the experiment. More specifically, the code to create nodes according to AWS specifications, create tasks according to Imdoukh paper, normalise fitness, select a solution from front, compare kubernetes with GA, and some visualisation scripts.

GitHub ID: tmathews75

### D. Kunduz Baryktabasova

Implemented Fitness Functions for both scheduling and rescheduling. Added Kubernetes algorithm for evolution. Wrote evolution for rescheduling. Worked on initial implementation of classes such as Node, Container, Chromosome.

GitHub ID: kunduzb17 and Kunduz Baryktabasova

The first half of the project Kunduz did commits from her PC, where Kunduz forgot to login to her github account, so commits were done by Kunduz Baryktabasova, the "commits" section can be checked(mostly the commits 1-50). After that Kunduz signed in to her account kunduzb17, and commits started to be done from this account.

## VI. Conclusion

In conclusion, as the result we were able to accomplish all the goals set: we implemented the Scheduling task which beats the performance of Kubernetes by 71% in some of the objectives and were the first to apply Search-Based technique on Rescheduling task. As the future work, the Scheduler and Rescheduler can be released as plug-in for Kubernetes.


[1] Fitness proportionate selection

[2] Guerrero, C., Lera, I. Juiz, C. *Genetic Algorithm for Multi-Objective Optimization of Container Allocation in Cloud Architecture*, J Grid Computing (2018) 16: 113.

[3] Mahmoud Imdoukh Imtiaz Ahmad Mohammad Alfailakawi *Optimizing scheduling decisions of container management tool using many-objective genetic algorithm*

[4] Kalyanmoy Deb, Himanshu Jain, *An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints*