EFFICIENT SCALES OF MICROSERVICE-ORIENTED SYSTEMS

A comparison of evolutionary algorithms

Daniel Harr

Master Thesis, 30 credits
Supervisor: Jerry Eriksson, Ahmed Ali-Eldin
External supervisor: Daniel Karlsson, Nasdaq
Civilingenjörsprogrammet i teknisk datavetenskap

2019
Abstract

Many modern software systems are designed into a microservice-oriented architecture as they run into issues when attempting to scale. An issue with large and complex microservice-oriented systems is to know which scales of a system that are well-performing with regard to resource usage. Identifying efficient scales is interesting to minimize resource usage and cost while maximizing performance.

The optimal scales of a demo system is investigated using multi-objective Ant Colony and Particle Swarm optimization. The optimization methods are evaluated and compared with respect to properties of the resulting set of scales, and how much of the search space that is discovered for the solutions to be produced.

The experiments show that Ant Colony is more consistent in producing the entire correct set of scales. Particle Swarm however is cheaper with regard to the number of scales that need to be tested in order to produce a result. Since testing a scale becomes more expensive as the investigated system grows in size and complexity, an initial conclusion is that Particle Swarm would be more viable for a real-world scenario. There are however some ideas of improvements that could affect the conclusions, and a larger and more complex system should be tested as well before any real conclusions can be made.
Acknowledgements

I am very grateful to Daniel Karlsson at the Minium project at Nasdaq, who had the initial project idea. I would also like to thank the people at Minium and Nasdaq in general for their support and providing me with a place to work. I am also grateful to Tobias Nebaeus whose help and collaboration has been invaluable in this thesis.

My sincere thanks to my supervisors Jerry Eriksson and Ahmed Ali-Eldin at Umeå University for their support during the project.

Last but not least I would like to thank family, girlfriend and friends for their motivation and support.
## Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACO</td>
<td>Ant Colony Optimization</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>GCP</td>
<td>Google Cloud Platform</td>
</tr>
<tr>
<td>GKE</td>
<td>Google Kubernetes Engine</td>
</tr>
<tr>
<td>MOACO</td>
<td>Multi Objective Ant Colony Optimization</td>
</tr>
<tr>
<td>MOPSO</td>
<td>Multi Objective Particle Swarm Optimization</td>
</tr>
<tr>
<td>PSO</td>
<td>Particle Swarm Optimization</td>
</tr>
</tbody>
</table>
3.3.4 Algorithm tests 16
3.4 Comparison of algorithms 16

4 Results 17
4.1 Size of the resulting Pareto set 17
4.2 Hypervolume indicators 18
4.3 Evaluated configurations 18
4.4 Evaluations 18
4.5 The Pareto set 19

5 Conclusions 23
5.1 Research questions 23
5.1.1 Comparison of number of solutions 23
5.1.2 Comparison of resource usage and evaluations 23
5.1.3 Comparison of hypervolume indicators 24
5.1.4 Comparison of produced Pareto sets 24
5.2 Goals 24
5.2.1 Comparison of complexity 24
5.2.2 Which algorithm fits Minium better? 25

6 Discussion 27
6.1 Changes to the test system 27
6.2 Changes to the algorithm metrics 27
6.3 Changes to the algorithms 28
6.3.1 Changes to MOACO 28
6.3.2 Changes to MOPSO 28
6.4 Code quality 28
6.5 Reflections 28

Bibliography 29
1 Introduction

A benefit of microservice-oriented systems is the ability to adapt and provision computer resources to different parts of the system at runtime [8]. This adaptive ability combined with a cloud environment allows for processing certain tasks faster, or running the system cheaper at a slower pace, at different times of day. It makes it easier to meet demand at peak hours without overprovisioning during calmer hours.

It can be difficult to allocate computer resources efficiently for complex microservice-oriented systems at large scale. Therefore it is interesting to investigate how evolutionary algorithms can be used to identify efficient allocations of these resources. This approach is targeted at discovering efficient scales beforehand for a predefined task. Knowing which scales performs well is useful for quick processing of scheduled tasks. The system can be scaled to quickly process the scheduled task and then return to its previous state.

This project studies methods of identifying optimal scales of microservice-oriented software systems with respect to resource usage and performance. The project is performed at the Minium project at Nasdaq aimed at their post-trade risk system.

This chapter introduces the concepts of microservices and optimization studied in this thesis, and how they can be applied at Minium. Then purpose and problem description are presented. Finally the research questions are presented along with limitations and related work.

1.1 Microservices

Microservices can be described as an architectural style for software systems. Traditional systems often consist of one large service. Microservice-oriented systems consists of several small services which work together, often distributed across several machines in a cloud environment. [8]

As systems need to increase in scale, they may hit barriers caused by software architecture. Microservice-oriented architecture removes some of these barriers in order to allow for easier scaling of the system. However, it does come with trade-offs. [8]

Some problems that surfaces with microservices is knowing and deciding when scaling is required and which services to scale. Finding efficient scale configurations for a microservice-oriented system can be a complex problem, and the complexity increases with the amount of services.

An example presenting different scale configurations is shown in Figure 1. The figure shows two different scale configurations for a system consisting of three services A, B and C. In scale configuration 1, service A, B and C are scaled to 3, 1 and 2 replicas respectively. In scale configuration 2, the services A, B and C are scaled to 2, 2 and 1 replicas respectively. A replica is a running instance of the service. Having three replicas means that there are three running instances of the service sharing the workload.
1.1.1 Microservices in practice

In this project, the microservices are run as Docker containers [10], which can be seen as a more lightweight alternative to virtual machines with the virtualization implemented at a different level. A container contains the application together with the necessary libraries. Figure 2 and 3 shows the difference in layer structure when running virtual machines compared to containers as described by Lechtenbörger [11].

The containers are deployed and orchestrated using a Kubernetes cluster [3]. Kubernetes automates the process of deploying and running container applications, and manages infrastructure such as computing, storage and network. Kubernetes also makes it simple to scale services. In this case the cluster is running in Google Kubernetes Engine [9], which provides...
1.2 Optimization

Multi-objective optimization methods usually produce several solutions. Each solution produced is optimal given at least one objective with respect to the other objectives. For example, the cheapest solution is probably not the best performing and vice versa. There may also be several solutions which are more or less cheap, but optimal given the performance they provide. The resulting set of solutions is called a Pareto set or a Pareto front. A Pareto set can be evaluated using the hypervolume indicator, which can be used to compare the quality of different sets.

Metaheuristic optimization methods do not consider problem-specific information, but rather the relationship between produced data. For example, it considers the cost difference between two solutions, but without any knowledge of how the costs were produced or what the values represent. Both algorithms studied in this thesis behave in this way.

Efficient scale configurations are interesting both from a performance and a resource utilization perspective. Since performance and resource usage often are conflicting, it does not necessarily exist a single solution. Using multi-objective optimization several near-optimal solutions can be discovered. Finding several efficient configurations can also aid identifying bottlenecks. The configurations could attempt to maximize performance or to minimize resource usage and cost.

1.3 Minium

Minium is a post-trade risk system targeting banks and brokers. The project has recently been acquired by and is being developed at Nasdaq. The Minium system is running with a microservice architecture in the cloud. With Kubernetes as the foundation, the system is built to be elastic, with most of the services being scalable. The incoming load to the system varies greatly during the day.

The system exposes an Application Programming Interface (API). Some API routes are only called during certain periods, and some services are not used at all for most of the day. The system is supposed to be up almost 24/7, but many hours will see a very small usage, while some shorter periods will receive massive amounts of data.

Since the system is scalable and they have an understanding of how the input to the system varies throughout the day, a solution regarding the scaling could be to have predetermined configurations that are applied at a specific point in time. Finding these configurations can however be tricky, since it is not known exactly how the input correlates to the load on the services.

Having an automated algorithm to find a configuration based on a predetermined input set is something that would simplify identifying these configurations, and there would be less guessing and trial and error.
1.4 Problem description

When gradually scaling a microservice-oriented system it is not always obvious which microservice to scale. The complexity increases with the amount of microservices and as they depend on other microservices. One way would be to scale all microservices simultaneously, but that would quickly become resource- and cost-inefficient.

The problem can be described as a combinatorial optimization problem with multiple objectives. These problems are usually difficult to solve, as they often are NP-hard. Metaheuristic methods utilize problem-independent information to find candidate solutions to a problem. Approximate methods using metaheuristics are often a feasible way of producing near-optimal solutions in relatively short time compared to exact methods. [6]

It is not uncommon for objectives to be in direct conflict with each other (high throughput versus low cost for example). Therefore multi-objective methods produce a set of candidate solutions (a Pareto front) from which a decision maker decides which solution to use. It is therefore interesting to find which candidate solutions have better trade-offs than others to aid decision making.

Different optimization methods fit some kinds of problems better than others. It is therefore interesting to compare different algorithms given this specific optimization problem.

1.5 Purpose

The purpose of this project is to compare different methods for multi-objective optimization when scaling a microservice-oriented system. Two multi-objective metaheuristic methods are applied on optimizing the amount of replicated services deployed in a system. The optimization is performed with respect to maximizing system performance and minimizing resource usage. The methods studied are Ant Colony [6][1] and Particle Swarm [7][5] optimization.

1.5.1 Goals

The goal of the project is to answer the following:

- How do these algorithms compare in terms of complexity?
  - Ease of understanding?
  - Ease of implementing?
- Which algorithm fits Minium better? Why?

Other goals

The algorithms should be implemented as a framework for multi-objective optimization. It should be easy to use the framework as long as the user provides the problem-specific code. The code should be readable and help in understanding the algorithms.
1.6 Research questions

The research questions in this project are:

- How do the algorithms compare in number of solutions produced?
- How do the algorithms compare in number of evaluations?
- How do the algorithms compare in terms of largest discovered hypervolume indicator?

1.7 Limitations

Computer hardware resources are limited. This limits the size and scale of software systems that can be used. Cloud resources are limited by the Google Cloud Platform Free Tier limitations, and the initial $300 credits.

Due to time taken to scale the system and provide relevant metrics, the size and complexity of the system is limited. Therefore only the most relevant services given a specific workload is scaled.

Due to the complexity of using distributed state, only the stateless services of a system are scaled.

1.8 Related work

Chen and Bahsoon [4] introduces an autoscaler using a MOACO inspired algorithm. They use compromise-dominance for identifying the non-dominated solutions with high-quality tradeoffs. The autoscaler is self-adaptive and attempts to reduce human involvement when identifying efficient tradeoffs.

Singh et al. [13] introduces a model for what-if analysis in complex cloud computing applications. This analysis allows for predicting application behavior after hypothetical changes based on monitoring information.
2 Theory

This chapter gives a more in-depth description of multi-objective optimization and the optimization algorithms studied in this thesis, Ant Colony Optimization and Particle Swarm Optimization.

2.1 Multi-objective optimization

Multi-objective optimization problems can be described as a set of optimization objectives, a set of decision variables, a search domain and a set of constraints. Solutions which are not optimal with regard to any objective are dominated. Solutions which are not dominated by any other solutions are non-dominated. Since objectives can be conflicting there usually exists more than one solution. The set of non-dominated solutions is called a Pareto set or front. [1]

Alaya et al. [1] describes this using buying a second-hand car as an example. It is desirable to buy a car in a good state at a low price. These objectives are usually conflicting. Car A is better than car B if A is in a better state at a lower price compared to B. If car A has better state than car B, but also higher price, they are not comparable. The decision variables here would be state and price. The search domain would consist of the state and price of all cars on the market. A constraint could be that the car should be roadworthy. If the buyer is on a budget, that could also be a constraint. By comparing all available cars to each other, only keeping the preferred and non-comparable ones, the Pareto set of solutions would be acquired. [1]

Figure 4 shows an example of minimizing the objectives \( f_1 \) and \( f_2 \) and a set of solutions. Solutions A and B together with the other solutions connected by dotted lines are the Pareto set. Solution C is dominated by both A and B and is therefore not part of the Pareto set.

![Figure 4: An example of a Pareto front or set minimizing the objectives \( f_1 \) and \( f_2 \). The solutions along the dotted line are the Pareto set. Solutions A and B are both non-dominated, but solution C is dominated by at least one solution.](image-url)

In this optimization problem the optimization objectives consist of system performance metrics at different system scales. The set of decision variables are the services that can be scaled, and the search domain restricting how the system can be scaled. A solution consists of a scaling configuration stating how many replicas there should exist for each microservice. Intuitively, the set of solutions should range from small-scale systems with few replicas and low performance, to large systems with high performance.

2.1.1 Evaluating multi-objective optimization methods

A performance metric for multi-objective optimization methods is the hypervolume indicator, or the S-metric. Given a minimization problem, the hypervolume indicator is the volume covered by the Pareto set and a limit above the maximum value of all objectives in objective space. The intersection of the limits of each objective results in a reference point. [2]

Figure 5 shows the hypervolume indicator given a reference point and the example shown in Figure 4.

![Figure 5: The hypervolume indicator computed for the Pareto front presented in Figure 4 given a reference point.](image)

2.2 Ant Colony Optimization

Dorigo and Stützle proposes Ant Colony Optimization (ACO) algorithms as an often well-performing method for solving combinatorial optimization problems. These algorithms are inspired by the group behavior of ant colonies as they coordinate and perform tasks such as “...foraging, division of labor, brood sorting and cooperative transport” [6 p. 1]. A main source of inspiration for ACO is how ants use pheromones to lay and follow trails to influence other ants. The pheromone-trail behavior allows ants to collectively discover food sources and short ways between the food source and their home. [6]

2.2.1 Multi-Objective Ant Colony Optimization

A Multi-Objective Ant Colony Optimization algorithm (MOACO) is proposed by Alaya, Solnon and Ghédira, which extends ACO algorithms with the possibility of providing multiple candidate solutions which take several objectives into consideration. [1]
López-Ibáñes and Stützle \[12\] proposes several variants of MOACO algorithms together with a framework for instantiating these algorithms. The framework describes the main algorithm of MOACO and is shown in Algorithm 1.

**Algorithm 1** Multi-objective Ant Colony Optimization as described by Lópe-Ibáñes and Stützle \[12\].

Initialize all pheromone trails to the maximum value

repeat
  for each colony do
    for each ant in colony do
      construct a solution
    for i in \# pheromone structures do
      update the ith pheromone structure trails
      if a trail is outside pheromone boundary values then
        set it to corresponding boundary value
  until maximum number of cycles reached

The solution construction algorithm is presented in Algorithm 2. The solution set $S$ is constructed from the candidate solution set $Cand$. The candidate set consists of possible solutions reachable from the current solution. The solutions are picked with a probability calculated by the following equation:

$$p^c_S(v_i) = \frac{[\tau^c_S(v_i)]^\alpha \times [\eta^c_S(v_i)]^\beta}{\sum_{v_j \in Cand}[\tau^c_S(v_j)]^\alpha \times [\eta^c_S(v_j)]^\beta}$$

where:

- $p^c_S(v_i)$ = probability of choosing candidate $v_i$ to solution set $S$ for colony $c$
- $\tau^c_S(v_i)$ = pheromone factor of candidate $v_i$
- $\eta^c_S(v_i)$ = heuristic factor of candidate $v_i$
- $\alpha$ = parameter that determine importance of pheromones
- $\beta$ = parameter that determine importance of heuristics

**Algorithm 2** Solution construction in MOACO

\begin{itemize}
  \item $S \leftarrow \emptyset$
  \item $Cand \leftarrow$ reachable solutions
  \item while $Cand \neq \emptyset$ do
    \begin{itemize}
      \item choose candidate $\in$ Cand with probability $p^c_S(candidate)$
      \item add candidate at the end of $S$
      \item remove candidates from Cand that violate constraints
    \end{itemize}
\end{itemize}

When updating the pheromone trails, the new pheromone values are calculated using the following equation:

$$\tau^i(c) \leftarrow (1 - \rho) \times \tau^i(c) + \Delta \tau^i(c)$$

where:

- $\tau^i(c)$ = pheromone value for the $i$th pheromone structure
- $\rho$ = evaporation factor, $0 \leq \rho \leq 1$
- $\Delta \tau^i(c)$ = the amount of pheromone laid on the structure by ants
2.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an optimization concept inspired by swarming theory (bird flocking, fish schooling) proposed by Eberhart and Kennedy. It is related to both genetic algorithms and evolution strategies. [7]

PSO works by initializing a population with random solutions and a randomized velocity. The individuals in the population, the particles, travel through the space of possible solutions, sharing information about possible optimal solutions with the population. The population-best solution is stored in a global repository. Population- and individual-best solutions together with current velocity and some random factors for acceleration dictates how the velocity changes for each individual. After letting the particles travel for a while, the population should have discovered a (near) optimal solution. [7]

2.3.1 Multi-Objective Particle Swarm Optimization

Coello and Lechuga proposed a Multi-Objective Particle Swarm Optimization extension (MOPSO) to solve multi-objective optimization problems. The extension consists of using a Pareto ranking scheme to store non-dominated solutions in the global repository, allowing particles to follow non-dominated solutions in search for optimal solution candidates. [5]

Non-dominated solutions are also represented in a hyperspace structure using their objective values as coordinates. The hyperspace is divided into hypercubes. When the particles are retrieving a population-best solution, they are more likely to get a solution from a less crowded hypercube. Coello and Lechuga considers this a form of fitness sharing. [5]

The main algorithm is presented in Algorithm 3. One of the more characteristic behaviors of this algorithm is how the particles are flying through hyperspace [7]. The velocity calculation is shown in the following equation:

$$v ← w × v + r_1 × (pbest − present) + r_2 × (gbest − present)$$

where:

- $v$ = velocity of the particle
- $w$ = inertia weight
- $r_1, r_2$ = random numbers between 0 and 1
- $pbest$ = best position visited by the particle
- $gbest$ = best position visited by the population
- $present$ = current position of the particle
Algorithm 3 Multi-objective Particle Swarm Optimization

initialize all particles
set all particles velocity to 0
evaluate all particles in the population
store positions of non-dominated solutions in the repository
generate hypercubes of search space explored so far
for all particles in the population do
  remember current position as the best position visited so far
while maximum number of cycles has not been reached do
  update velocity for all particles
  update position for all particles
  maintain particles within search space
  evaluate all particles
  update repository and hypercube representation
for all particles in the population do
  if a particle’s new position is better than the best visited so far then
    update best position for the particle
3 Method

This chapter describes the tasks required to answer the research questions.

3.1 Literature study

The concept of multi-objective optimization and the algorithms to use are studied. Cloud platforms and methods of running microservice-oriented systems are studied as well.

3.2 Implementation of algorithms

Both algorithms are implemented according to relevant articles read in the literature study. They are based on the algorithms presented in the Theory chapter. Many parameter values in the implementations are taken from literature for simplicity, but the algorithms could be optimized to find well-performing values for this specific problem.

The algorithms are implemented in Python 3.

3.2.1 Multi-objective Ant Colony implementation

The implemented algorithm is more simple compared to the framework presented in Algorithm 1. There is only one colony, and therefore no colony-related structure is implemented.

The solutions are represented as scale configurations. When creating a solution, the solution algorithm presented in Algorithm 2 is simplified to choose a candidate based on probability, rather than constructing a solution. The concept of constructing a solution as described in the algorithm is not applicable to the problem studied in this thesis. The candidate solution set consists of the solutions differing by one replica while remaining within the search space, given a previous solution.

The parameters used for MOACO is shown in Table 1. \( \alpha \) and \( \beta \) is the relative importance of pheromones and heuristics respectively when creating solutions. There is no hard limit on how many solutions that can be produced. When updating the pheromone trails, only the globally best solutions are awarded with pheromones.

3.2.2 Multi-objective Particle Swarm implementation

The Particle Swarm Optimization implementation resembles the algorithm presented in Algorithm 3. However, the algorithm is adjusted to output discrete solution values, since it works with floating point values by nature. This is done by rounding the values in the solutions towards zero before returning them. Note that the algorithm still uses floating point values internally.
### Table 1 Parameters for MOACO.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>Varies by test</td>
</tr>
<tr>
<td>$\alpha$</td>
<td>0.5</td>
</tr>
<tr>
<td>$\beta$</td>
<td>0.5</td>
</tr>
<tr>
<td>Maximum pheromone value</td>
<td>100</td>
</tr>
<tr>
<td>Minimum pheromone value</td>
<td>0</td>
</tr>
<tr>
<td>Pheromone reward</td>
<td>10</td>
</tr>
<tr>
<td>Evaporation rate</td>
<td>0.1</td>
</tr>
</tbody>
</table>

The parameters used for MOPSO is shown in Table 2. Since the problem is discrete of nature and the search space is limited, the memory footprint of the repository should not be an issue. Therefore the repository size is set to 1000 which in effect should behave as if it is unlimited in these tests. This also behaves like MOACO which does not have a limit on how many solutions that can be produced.

### Table 2 Parameters for MOPSO.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Population size</td>
<td>Varies by test</td>
</tr>
<tr>
<td>Inertia weight</td>
<td>0.4</td>
</tr>
<tr>
<td>Repository size</td>
<td>1000</td>
</tr>
<tr>
<td>Hypercubes per dimension</td>
<td>5</td>
</tr>
</tbody>
</table>

### 3.3 Experiments

Experiments are made in order to produce results. The experiment environment consists of several parts. One part for gathering system metrics such as performance and resource usage, and one part for running the algorithms on the produced data.

To produce data for analyzing, a cloud environment and a target system is required. In order to analyze performance a tool for applying load on the target system is also required. The system and the artificial workload should be simple enough to provide predictable results.

Figure 6 shows the parts of the experiment environment which are individually described in the following sections.

![Figure 6: The layout of the experiment environment.](image)
3.3.1 Target system

The demo system is deployed as Docker containers in a Kubernetes cluster on GKE. The cluster consists of 3 n1-highcpu-2 virtual machines, each having 2 vCPUs and 1.8 GB memory.

The system consists of seven services and is visualized in Figure 7. All services are scalable. A client sends requests to the system, which ends up in an Entry service. The Entry service sends a request to the services A, B and C each. Service A sends a request to service 1 and 2 each. Services B and C both sends a request to service 3.

The services 1, 2 and 3 calculate a Fibonacci sequence to simulate a workload. Service 1 calculates the sequence one time per request. Service 2 calculates the sequence two times per request. Service 3 calculates the sequence four times per request.

![Figure 7: The target system.](image)

3.3.2 Load tool

The load tool is a Python script which sends a fixed amount of requests concurrently. The tool presents the amount of queried requests along with the time taken for each request, which makes it possible to analyze response times. There is only one kind of request considered in this test. The tool is running on a virtual machine of type n1-highcpu-2 having 2 vCPUs and 1.8 GB memory. The virtual machine is running in the same zone as the cluster which hosts the target system.

3.3.3 System metric gathering

As the algorithm requests information for a configuration, the target system scales to that configuration and tests are performed to provide the requested metrics. The test results are stored to a file so that each configuration only has to be tested one time per test environment. As an attempt to maintain consistent results, the results should come from the same setup of virtual machines, i.e. without restarting any virtual machines.

The boundaries on the search space is shown in Table 3 and results in a search space of 10125
The true Pareto set given the system metrics is computed by removing all Pareto dominated solutions from the data set.

### Table 3 Minimum and maximum number of instances for each service.

<table>
<thead>
<tr>
<th>Service</th>
<th>Minimum</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Service A</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Service B</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Service C</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>Service 1</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Service 2</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Service 3</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>

#### 3.3.4 Algorithm tests

To be able to evaluate the algorithms and compare them to each other, they are tested several times on different population sizes (particles and ants for PSO and ACO respectively) and over different amounts of iterations. The population sizes are 50 and 100 individuals. The iterations are 10, 50, 100, 500 and 1000 iterations. Each test is repeated 20 times.

A requirement for the tests is the hypervolume reference point. In this case, the only unknown value is the average response time. The reference value has been chosen after the whole search space has been explored, and then set to a value higher than the highest average response time. Maximum memory and CPU is known beforehand since there are limits on the maximum scale of the system.

A test result consists of a Pareto front with scale configurations along with their objective values. Hypervolume values for each iteration is also available. The experiments also notes metrics from the algorithms such as evaluations and iterations performed.

#### 3.4 Comparison of algorithms

The algorithms are compared based on the results of the experiments. The comparison of the experiment results should present enough to answer the research questions and draw conclusions regarding the goals.
4 Results

This chapter presents the results produced by running the experiments as described in the Method chapter. All services reserve the same amount of resources and therefore the resource usage objectives are simplified from memory and CPU usage to total number of service instances.

4.1 Size of the resulting Pareto set

Figure 8 shows the sizes of the resulting Pareto sets for both algorithms. The algorithms are compared at different population sizes and iterations per test run.

![Graph showing the size of the resulting Pareto set](image)

**Figure 8:** The solution count per test run for both algorithms with different population sizes and iterations.
4.2 Hypervolume indicators

Figure 9 shows the largest discovered hypervolume indicators for both algorithms. MOPSO discovered the largest hypervolume once. MOACO discovered the largest hypervolume 63 times.

![Largest discovered hypervolume indicators](image)

**Figure 9:** Largest discovered hypervolume per test run for both algorithms with different population sizes and iterations.

4.3 Evaluated configurations

Figure 10 shows the amount of explored configurations for both algorithms. The search space consists of 10125 different configurations.

4.4 Evaluations

Figure 11 shows the amount of evaluations done for both algorithms at different population sizes and iterations.
Figure 10: Evaluated configurations per test run for both algorithms with different population sizes and iterations.

4.5 The Pareto set

A Pareto set produced by removing dominated solutions from the system metrics is shown in Table 4. Note that this is not produced by running the algorithms.

The Pareto sets with highest hypervolume indicator for both algorithms together with the computed Pareto set is shown in Figure 12.
Figure 11: Evaluations per test run for both algorithms with different population sizes and iterations.
Table 4 The configurations in the computed Pareto set given the system metrics.

<table>
<thead>
<tr>
<th>Entry</th>
<th>Service A</th>
<th>Service B</th>
<th>Service C</th>
<th>Service 1</th>
<th>Service 2</th>
<th>Service 3</th>
<th>Average response time (s)</th>
<th>Memory (MB)</th>
<th>CPU (mCPU)</th>
<th># services</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2.54</td>
<td>350</td>
<td>700</td>
<td>7</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1.50</td>
<td>400</td>
<td>800</td>
<td>8</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1.24</td>
<td>450</td>
<td>900</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1.15</td>
<td>500</td>
<td>1000</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>1.08</td>
<td>550</td>
<td>1100</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>4</td>
<td>0.99</td>
<td>600</td>
<td>1200</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>5</td>
<td>0.88</td>
<td>650</td>
<td>1300</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>5</td>
<td>0.86</td>
<td>700</td>
<td>1400</td>
<td>14</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>0.84</td>
<td>750</td>
<td>1500</td>
<td>15</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>5</td>
<td>0.74</td>
<td>800</td>
<td>1600</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>5</td>
<td>0.73</td>
<td>900</td>
<td>1800</td>
<td>18</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>0.70</td>
<td>1250</td>
<td>2500</td>
<td>25</td>
</tr>
</tbody>
</table>
Figure 12: Overlapping evaluated and computed Pareto fronts.
5 Conclusions

This chapter presents conclusions made given the results presented in Chapter 4 regarding the research questions and goals.

5.1 Research questions

This section covers conclusions made regarding the research questions.

5.1.1 Comparison of number of solutions

As seen in Table 4 and Figure 12, the optimal Pareto set given this search space contains 12 solutions. Figure 8 indicates that MOACO tends to produce a set containing 12 solutions more often than MOPSO.

Since MOACO evaluates far more configurations (see Figure 10 and 11) and the ants movement is largely based on the heuristics of neighboring configurations, their movement behavior may be beneficial comparing to the particles in MOPSO with regards to identifying the Pareto set. The particles do usually not know anything regarding the heuristics of the neighbouring configurations, and would not take it into consideration unless it is a population-best or individual-best configuration, which could lead to less efficient movement behavior compared to MOACO. However, ants still need to evaluate all neighbors when moving, leading to expensive movement in unexplored areas of the search space, which could imply that MOPSO is more efficient with regards to the cost of an evaluation.

MOPSO would probably perform better if it had the extreme values evaluated beforehand and added to the global repository of population-best solutions, since it then would not have to discover all edge configurations.

5.1.2 Comparison of resource usage and evaluations

Figure 10 shows how many configurations that are evaluated for both algorithms. MOACO tends to evaluate the entire search space for the larger tests. That is very much compared to MOPSO which only evaluates more than 2000 evaluations in one test case. This is a substantial difference of a factor of almost 5 for the worst case of MOPSO compared to 3 cases of MOACO which evaluated the entire search space in all 20 test runs.

Evaluations are considered a metric of resource usage since it inflates the cache with regards to memory usage and the evaluations make up a substantial amount of the total test time until the whole search space has been sufficiently evaluated.
5.1.3 Comparison of hypervolume indicators

Figure 9 indicates that MOPSO converges faster than MOACO since the hypervolume indicators are higher than for MOACO for lower iterations. However, MOACO seems to be more consistent with the result for iterations at 500 and 1000. MOACO also manages to discover superior Pareto sets much more frequently. This is consistent with Figure 8 which indicates that MOPSO usually misses one solution in the set in the larger tests.

5.1.4 Comparison of produced Pareto sets

Figure 12 shows that both algorithms can produce the true Pareto set. Both algorithms discover sets of size 12 according to Figure 8 which could support that conclusion. However, as also shown in Figure 8, MOPSO does not ever have size 12 as average, whereas MOACO produces a set of size 12 on average for 4 test runs. Also, MOPSO only found the largest discovered hypervolume once compared to 63 for MOACO. This indicates that it is far more likely for MOACO to discover the entire true Pareto set in a test run.

5.2 Goals

This section covers conclusions made regarding the goals.

5.2.1 Comparison of complexity

The algorithms are compared regarding how complex they are to understand and how complex they are to implement.

Ease of understanding

The concepts of both algorithms are quite similar. A population of individuals moving around looking for optimal solutions, sharing information with the population in some way. MOPSO is pretty straightforward, each particle has a position and calculates their velocities based on the personal-best and population-best solutions. MOACO has a more complex mechanism, where the population-best solutions are rewarded with higher pheromone values which increases the probability that an ant will move there later. The movement for ants is probability based, weighted by pheromones and heuristics, which is not quite as straightforward as the behaviour in MOPSO.

Ease of implementation

The base algorithm of MOPSO is more simple than MOACO, but with the simplifications made to MOACO the difficulty is quite similar. Many of the multi objective optimization parts are common and is shared between algorithms since they use the same representation of the problem. The most difficult part of MOPSO is to implement the adaptive grid for representing the hypercubes, but is otherwise very straightforward to implement.

There are more decisions to be made when implementing MOACO, such as which solutions are candidates when ants are moving and how to use the pheromone values. There are also more parameters to set when using MOACO. Due to this, MOPSO is more simple to implement
and use.

5.2.2 Which algorithm fits Minium better?

Considering the difference in evaluations, MOPSO is probably the superior choice for Minium. Especially when also considering that the tests for gathering metrics in this case probably takes less time compared to when testing a real system with larger workloads. As the time taken for each evaluation increases, the more MOPSO benefits. However, if the search space is small or evaluations are cheap, MOACO seem to produce more consistent results and more often the true Pareto set.
6 Discussion

This chapter contains discussion such as reflection of the project, and interesting changes and possible improvements to the work that has been done. That includes some suggestions for future work.

6.1 Changes to the test system

The test system had to be kept very basic due to limitations on time and money. It would be interesting to apply this to a real system and see what happens. It would also be interesting to put more effort into creating a more realistic workload simulation, both how it is generated and its constituents.

Testing the algorithms with a system at a much larger scale, both in number of distinct services and number of instances, would be interesting in order to compare how the algorithms scale. This would almost certainly rather quickly lead to issues in MOACO and require measures such as reducing the set of candidates to a smaller set by some measure, or the number of evaluations by using predictions and estimates with what-if analysis.

Some attempts have been made to keep the measurements of the system metrics consistent and isolated to the microservices, such as not restarting the virtual machines. However, they are not in my control and GCP could easily have migrated the virtual machines around to different machines without my knowledge, possibly leading to side effects in the results. One solution to reduce sensitivity to events like this could be to collect metrics more than once for each scale configuration, and at different moments in time. The collected metrics could then be processed some way, such as averaging or by taking the minimum, before used in the algorithms. This should reduce the impact of other elements than the performance of the system that is being investigated.

6.2 Changes to the algorithm metrics

In the cases when the true Pareto set is known, it would be interesting to compare the number of iterations taken to find the true Pareto set. This metric could become a good objective for optimizing the parameters of the algorithm: minimize average number of iterations needed to find the optimal set while tuning the algorithm to be as cheap as possible.

In this thesis a lot of the comparisons are based on number of iterations, or steps for each individual. Since the algorithms behave so differently in that respect, it could be interesting to have comparisons more based on number of evaluations.
6.3 Changes to the algorithms

There are some changes and improvements that could be done to the algorithms.

A change that could improve the overall results of the algorithms is using compromise-dominance, as it would filter out solutions with inferior tradeoffs. The algorithms could also be optimized by themselves or other optimization methods to identify well performing parameter values.

6.3.1 Changes to MOACO

The pheromone reward system is very simple at the moment and there is probably plenty of work that can be done to improve it. The maximum and minimum pheromone values, the pheromone reward and the evaporation rate are parameters that can be tuned to promote different behavior for the ants. The pheromone reward could also be based on some logic rather than the fixed value that is used in this implementation.

The effect of rewarding neighbors to non-dominated solutions could be investigated, as it would let ants know that they are getting closer to a well-performing part of the search domain. Currently the pheromone values only carry any effect if an ant is standing on a position that is next to a non-dominated solution.

6.3.2 Changes to MOPSO

The inertia weight for the particles is only set to a value in this implementation. The algorithm could possibly be improved by investigating different values, or increasing or decreasing the value over time. The same goes for how the hypercube space is divided.

6.4 Code quality

Since some parts were complex to implement, there may be bugs in the algorithms which affects the results. Initially there was an attempt to implement the MOACO framework presented in [12], but after spending some time on it focus was shifted to implement a more specialized model for this problem. The implementations are not as generic as I would have wanted, but I am satisfied with the results nevertheless.

Both algorithms could benefit from parallelization, especially at larger population sizes when running on cached evaluations.

6.5 Reflections

Since MOACO individuals perform more evaluations per step, MOPSO could easily run with larger population size compared to MOACO in practice.

Both algorithms could probably benefit from having knowledge of the edge configurations at the beginning, such as the largest and the smallest scale of the system in the search space. Intuitively, at least the smallest scale should be part of the Pareto set since it is the cheapest.
Bibliography


