National College of Ireland

A Comparative Study of Metaheuristic Algorithms for Enhancing Topology-Aware Scheduling in Kubernetes

MSc Research Project
Cloud Computing

Harsh Harendra Singh Mall
Student ID: 21223572

School of Computing
National College of Ireland

Supervisor: Prof. Diego Lugones

National College of Ireland
Project Submission Sheet
School of Computing

Student Name:	Harsh Harendra Singh Mall
Student ID:	21223572
Programme:	Msc. Cloud Computing
Year:	2023
Module:	MSc Research Project
Supervisor:	Prof. Diego Lugones
Submission Due Date:	20/12/2018
Project Title:	A Comparative Study of Metaheuristic Algorithms for Enhancing Topology-Aware Scheduling in Kubernetes
Word Count:	6615
Page Count:	21

I hereby certify that the information contained in this (my submission) is information pertaining to research I conducted for this project. All information other than my own contribution will be fully referenced and listed in the relevant bibliography section at the rear of the project.

ALL internet material must be referenced in the bibliography section. Students are required to use the Referencing Standard specified in the report template. To use other author's written or electronic work is illegal (plagiarism) and may result in disciplinary action.

Signature:	Harsh Harendra Singh Mall
Date:	18th September 2023

PLEASE READ THE FOLLOWING INSTRUCTIONS AND CHECKLIST:

Attach a completed copy of this sheet to each project (including multiple copies).	☐
Attach a Moodle submission receipt of the online project submission, to each project (including multiple copies).	☐
You must ensure that you retain a HARD COPY of the project, both for your own reference and in case a project is lost or mislaid. It is not sufficient to keep a copy on computer.	☐

Assignments that are submitted to the Programme Coordinator office must be placed into the assignment box located outside the office.

Office Use Only	
Signature:	
Date:	
Penalty Applied (if applicable):	

A Comparative Study of Metaheuristic Algorithms for Enhancing Topology-Aware Scheduling in Kubernetes

Harsh Harendra Singh Mall
x21223572

Abstract

Containerization has become a game-changing technology that completely transforms how we package software applications. And when it comes to managing and scaling these containerized apps in the cloud, effective container orchestration becomes absolutely crucial. Kubernetes, a widely recognized platform, excels in orchestrating the deployment, scaling, and management of containers. However, the proposed solution of default scheduling method employed by Kubernetes for allocating containers to suitable nodes within the datacenter is primarily optimized for cloud workloads, , but unfortunately does not consider topological information about distributed environments int the context of fog and edge computing.

In this research, we propose a comparative analysis of three metaheuristic algorithms applied to Kubernetes to achieve topology aware scheduling. Our investigation centers around crucial metrics like execution time and the cost when the topology information is considered when scheduling containers. Experiments show that Genetic Algorithms has the lowest overall execution time and Local Search Algorithm is better cost effective. We observe a latency improvement of Genetic Algorithm as compared to Local Search Algorithm and a cost reduction of Local Search algorithm when compared to Genetic and Tabu Search Algorithms.

1 Introduction

The introduction of cloud computing has completely changed the computing landscape and offers enormous advantages for businesses and organizations all over the world. Access to a flexible and scalable pool of computer assets, including networks, servers, storage, and applications, is made possible by cloud computing. Its significance cannot be understated since it gives enterprises the opportunity to scale, be cost-effective, be agile, and have simpler administration. With the cloud, companies can expand and deploy apps quickly, save money on infrastructure, and concentrate on their core capabilities.

Parallel to this, containerization has arisen as a potent technology that enables the deployment of software applications and their dependencies into independent units known as containers. Applications execute consistently across a range of computer settings due to containers' lightweight and uniform run-time environments.

Container orchestration is a an important component of managing containers in cloud. Kubernetes is a well-known participant in this market, and it entails orchestrating the deployment, scaling, and administration of containers. The most widely used and accepted container orchestration platform is known as Kubernetes. The deployment, scaling,

load balancing, and self-healing of containers are all automated by Kubernetes, resulting in excellent availability and effective resource use.

In this section, we get into the importance of container scheduling in Kubernetes and its performance in field of fog and edge computing which is limited by the fact that topological information of the network is not considered. In this report, we explore different meta-heuristic algorithms that consider topology information and provide the Kubernetes scheduler with location optimized scheduling information for deploying containers.

1.1 Importance of Container Scheduling

Recent years have seen a revolution in software development and deployment thanks to the development of container technology, which was pioneered by the widely used Docker platform. A revolutionary method for packaging and executing applications, containers offer a consistent and dependable run-time environment across a range of computer systems. Containers are crucial because they speed up the software development and deployment processes. Containers make the deployment of programs across many contexts simpler by eliminating conflicts and incompatibilities by encapsulating all essential dependencies and parameters. They facilitate scalability, allowing programs to grow and reproduce themselves without interruption in response to demand. Due to its portability and small weight, Docker in particular has become quite popular in speeding up the development process, and enabling the widespread use of microservices architecture.

But as containerized applications increase in size and complexity, it becomes increasingly important to have effective container orchestration. Managing many containers while performing operations like deployment, scaling, load balancing, and fault tolerance is known as container orchestration. Among the different systems for container orchestrations like Docker Swarm, Kubernetes and Apache Mesos, Kubernetes has become the norm in this sector.

In the realm of Kubernetes, the work of allocating containers to the most appropriate nodes within a cluster is handled mostly by the default scheduling method. It is properly referred to as the "default scheduler," is essential for resource allocation optimization and efficient utilization of the existing computer resources. It considers a number of variables, such as the particular resource needs of containers, any outlined affinity or anti-affinity rules, and other restrictions established by users or the system itself. The default scheduler works to evenly distribute tasks throughout the cluster in a way that enhances performance and guarantees the general stability of the entire system by carefully weighing these factors. It's crucial to understand, though, that in fog and edge computing, the Kubernetes' built-in scheduling method presents multiple limitations.

Authors Rejiba and Chamanara (2022), delve into the realm of custom schedulers employed across various domains. Their work addresses the default scheduler's limitations, particularly in the context of fog and edge computing. The authors mention that topological awareness of fog nodes is one of the areas that requires development. They draw attention to the fact that in circumstances involving fog and edge computing, the default scheduler is unable to effectively manage resource allocation and task distribution. This emphasizes the demand for special scheduling algorithms created to handle the unique problems in these distributed and resource-constrained environments.

1.2 Research Objective

Considering the limitations of the default scheduler in Kubernetes, particularly in relation to the topological information of fog and edge computing environments, this research delves into the scheduling aspect of the orchestration tool, specifically tailored to the unique requirements of fog and edge scenarios. While there are various approaches to developing a custom algorithm, the meta-heuristic and hybrid meta-heuristic approaches have emerged as the most commonly used algorithm in this context, which brings us to our research question -

How does the performance and efficiency of Kubernetes in scheduling improve when applying topology-aware metaheuristic algorithms, and how do these algorithms compare in terms of execution time and cost?

The objective of this research is to create and implement a framework which provides details about the network latencies between fog nodes and to implement and compare three different metaheuristic algorithms on the basis of execution time and cost.

1.3 Structure and Outline

The remaining sections of this thesis are divided into five sections as described below

- Section 2 presents the related works that have been implemented related to the topic of kubernetes scheduling in the field of fog and edge.
- Section 3 of this paper details about the setup and the methodology followed
- Section 4 provides an overview of the design and implementation of this work and its details.
- Section 5 provides the experiments carried out and its results.
- Section 6 is the final section which provides the conclusion and future work.

2 Related Work

In this section, most of the paper deals with fog and edge computing scenarios in custom scheduling. As we know in fog and edge all the nodes are situated in different geographical locations, it is important to know the topology of the nodes, for better communication and to improve costs.

In the paper (Santos et al.; 2019), the filtering step of the default Kubernetes scheduler is extended by the authors' topology-aware scheduler. Their strategy comprises optimising the network distance between a particular Pod and a target location defined in its configuration file by employing round-trip time labels, which are allocated to cluster nodes in advance. The round-trip time labels are issued statically to cluster nodes, which means they do not take into account dynamic fluctuations in network latencies that may occur during run-time, which is one shortcoming of this technique. Due to the possibility that the static labels might not precisely reflect the state of the topology, this can result in less-than-ideal scheduling decisions.

In their research, authors (Kayal; 2020) address the issue of fog nodes' finite resources, which might result in a lack of capacity for multi-container pod hosting. They provide a

unique solution to this problem that entails dividing such pods into their component containers and spreading them among several fog nodes. As a result, a topology-aware node scoring system is presented to assess the appropriateness of fog nodes for each container. This ranking takes the distances between the various fog nodes into consideration and helps decide where to position each container for best performance and shortest communication latency. It should be noted that while though the authors refer to these actions and ideas as scheduling plugins, they haven't given any specifics on how to carry them out, as it is only a theoretical framework.

The authors (Fu et al.; 2021) present Nautilus, a dynamic run-time system that integrates a number of modules, one of which is a communication-aware microservice mapper. Based on the communication overhead between microservices, this particular module is in charge of dividing the microservice graph into several pieces. In order to ensure frequent data exchanges take place in memory, it is important to effectively map these divisions to cluster nodes.

Although Nautilus provides a way to move application pods when there is an uneven distribution of computing resources among nodes, one drawback is that it does not provide Pod rescheduling in situations when connectivity between microservices has been compromised. In situations where microservice interactions are disrupted, the system might not be able to dynamically adjust to changes in communication patterns, which could result in less-than-ideal performance.

The authors (Nakanishi et al.; 2020) present a novel approach to improve Kubernetes utilisation over a wide area network (WAN), with an emphasis on deployment of applications at the network edge. They make adjustments to the Kubernetes scheduler, which now takes the Autonomous System (AS) path of the Border Gateway Protocol (BGP) into account. This indicates that when choosing an AS path for deployment, the scheduler prioritises shorter AS pathways. The results of the trials show that this strategy reduces application access times, which is essential in edge contexts.

The issue of higher latencies brought on by users receiving service from far-off fog nodes is covered in this paper (Nguyen et al.; 2020). To accommodate increasing demand at particular fog node locations, Kubernetes scheduling algorithm might not create new replicas. The authors suggest associating nodes with their individual locations in order to overcome this issue and allowing users to indicate their preferred deployment site in the pod specification, with varying priority if necessary. The evaluation's findings show that this strategy successfully lowers latencies for users in areas where client requests are increasing.

The authors (Eidenbenz et al.; 2020) concentrate on reducing application latency to optimise industrial automation applications. They do this by transforming the application placement issue in a Kubernetes-based fog computing environment as an optimisation effort with application delay as the primary cost parameter, with the goal of minimising expenses. Their suggested solution offers an estimated approach that accounts for a number of variables, including location restrictions, linkages between fog nodes, links between application components, and their respective data requirements. The authors conduct experiments to show that the best outcomes are obtained when their technique is implemented utilising native Kubernetes capabilities like priority classes and pod affinities. Using this method, the deployment of industrial automation applications is successfully optimised, resulting in decreased latency and enhanced performance in the fog computing environment.

In paper (Haja et al.; 2019), the authors construct scheduling algorithms that takes

the topology of edge nodes into account in order to enable latency-sensitive edge applications.They provide a unique strategy that uses repeated measurements of edge node delays to accomplish this. These metrics are used to categorise the nodes according to their individual delays. The suggested scheduler for a pod that contains a delay restriction verifies the labels that have been assigned to each node. In order to ensure that latency-sensitive applications are deployed on the most suitable edge nodes, the scheduler then chooses the node that can satisfy the stated delay requirement.

In paper (Rossi et al.; 2020),the authors presented "ge-kube," a specialised version of Kubernetes created specifically for geographically dispersed installations. Two major contributions are highlighted in the study. To improve scalability and resource management, they first use reinforcement learning to figure out the ideal number of application clones. Second, in order to reduce deployment time and resource allocation, their strategy focuses on the effective placement of application instances. Their method does this by accounting for both the network delays between nodes and the resources that are available on each node. Ge-kube guarantees that application instances are positioned in a way that shortens deployment time and optimises resource utilisation in a geo-distributed environment by taking these aspects into account simultaneously.

In this paper (Marchese and Tomarchio; 2022), the authors provide a topology-aware scheduler scoring plugin and a unique descheduler operator created specifically for Kubernetes. The score plugin considers a number of variables, including the state of the cluster network, node-to-node latencies, and the traffic transferred between microservices in distributed applications. They use a Prometheus metrics server, where network latencies are assessed using node probes and application traffic data is gathered using Envoy proxies operating within each application Pod, to retrieve the pertinent metrics. The descheduler periodically decides to evict Pods for better placement while continually monitoring the network circumstances and application load.

The authors illustrate the efficacy of ge-kube through tests utilising a geo-distributed Redis cluster(database). It outperforms the built-in Kubernetes scheduler, increasing operations per second by an impressive 3 times. These results show how ge-kube significantly enhances the performance of geo-distributed deployments, making it a potential option for effectively handling such situations.

In the paper (Pusztai et al.; 2021), authors propose Pogonip a specialised edge-aware scheduler created for handling asynchronous microservices in Kubernetes. They solve the placement issue by modelling it as an optimisation problem for integer linear programming. They provide a heuristic strategy, which acts as an approximation solution method, to effectively uncover practical answers for real-world circumstances. A collection of scheduler plugins for Kubernetes are used to accomplish this heuristic.

Pogonip's significant drawback is that it does not provide pod rescheduling in response to evolving network circumstances. This indicates that the system might not dynamically adjust to changing communication patterns, which could result in performance issues when network circumstances change.

(Caminero and Muñoz-Mansilla; 2021) describes an addition to the standard Kubernetes scheduler that uses network status data to identify the best fog node for completing a batch job application within a certain time frame. To determine if an application can be finished before the deadline, the scheduler includes a prediction function. The scheduler rejects the application if it decides that the deadline cannot be reached.

It is significant to note that the suggested method does not take into account communication exchanges between Pods and is primarily intended for batch processes. As a

result, it might not be the best option for instances where inter-microservice communication is crucial. Table 1 provides a summary of limitations in the related work described in this section.

Paper	Approach and Contribution	Limitations
(Santos et al.; 2019)	Extended Kubernetes scheduler with topology-aware strategy using RTT labels for Pod placement.	Static labels ignore dynamic network latency fluctuations.
(Kayal; 2020)	Dividing multi-container pods among fog nodes based on topology-aware node scoring.	Lack of specifics for implementing theoretical framework.
(Fu et al.; 2021)	Nautilus: Communication-aware microservice mapper, dynamic run-time system.	Inability to reschedule Pods after microservice connectivity disruption.
(Nakanishi et al.; 2020)	Kubernetes scheduler considering Autonomous System (AS) path of BGP for edge deployment.	Focus on edge deployment, may not address all scenarios.
(Nguyen et al.; 2020)	User-defined preferred deployment site, associating nodes with locations to reduce latencies.	Limited to addressing latencies caused by distant nodes.
(Eidenbenz et al.; 2020)	Optimizing industrial automation app latency using Kubernetes capabilities.	Focus on specific use case, may not generalize well.
(Haja et al.; 2019)	Scheduling algorithm considering edge node topology for latency-sensitive apps.	Limited to latency-sensitive applications.
(Rossi et al.; 2020)	Ge-kube: Reinforcement learning for scaling and effective placement in geo-distributed environments.	Specific to geo-distributed deployments.
(Marchese and Tomarchio; 2022)	Network-aware scheduler and descheduler for Kubernetes with consideration of network metrics.	Network-centric, may not cover all scenarios.
(Pusztai et al.; 2021)	Pogonip: Edge-aware scheduler using heuristic approach for microservices in Kubernetes.	No pod rescheduling for changing network circumstances.
(Caminero and Muñoz-Mansilla; 2021)	Network-based fog node selection for batch job completion.	Primarily for batch processes, may not fit all cases.

Table 1: Summary of Research Papers on Kubernetes Topology Awareness

3 Methodology

In the previous section, we have discussed the different approaches taken to address the limitations of the default scheduler that Kubernetes offers. Kubernetes is built for large scale applications and is tailored for data centers in which network performance is rather homogeneous and consistent. However, it has its limitations in the context of fog where

the edges are situated in different geographical locations, possibly in different countries. In such topologies, network latencies play an important role, where it can range from 100ms to 400ms. Such delays are very significant, hence, consideration of topology is crucial when scheduling containers in Kubernetes. In this section, we are going to look at a methodology to simulate these environment while considering algorithms that take topology into considerations and with software and tools.

3.1 IfogSim Simulation Process

IfogSim is a powerful toolkit used for monitoring and various different tasks like resource management in Internet of Things, Fog and Edge computing environments with microservice orchestration and dynamic clustering as its features developed by Perez Abreu et al. (2019). In this paper, all the simulation is done in iFogSim. Ifogsim is an extension of CloudSim which was created by author Calheiros et al. (2011). The architecture of the simulation process is given in Figure 1. iFogSim offers a variety of classes designed to make simulation easier. These classes simulate various different components like a data center, fog nodes, application and cloudlet. The fog device class contains the core of fog nodes which includes important characteristics like processing power, energy efficiency and communication abilities. The specialised Cloudlet class, represents small fog nodes that operate as intermediaries between users and fog nodes. Cloudlets are represented as containers. It has characteristics, resource capacity, and communication profiles tailored to each place.

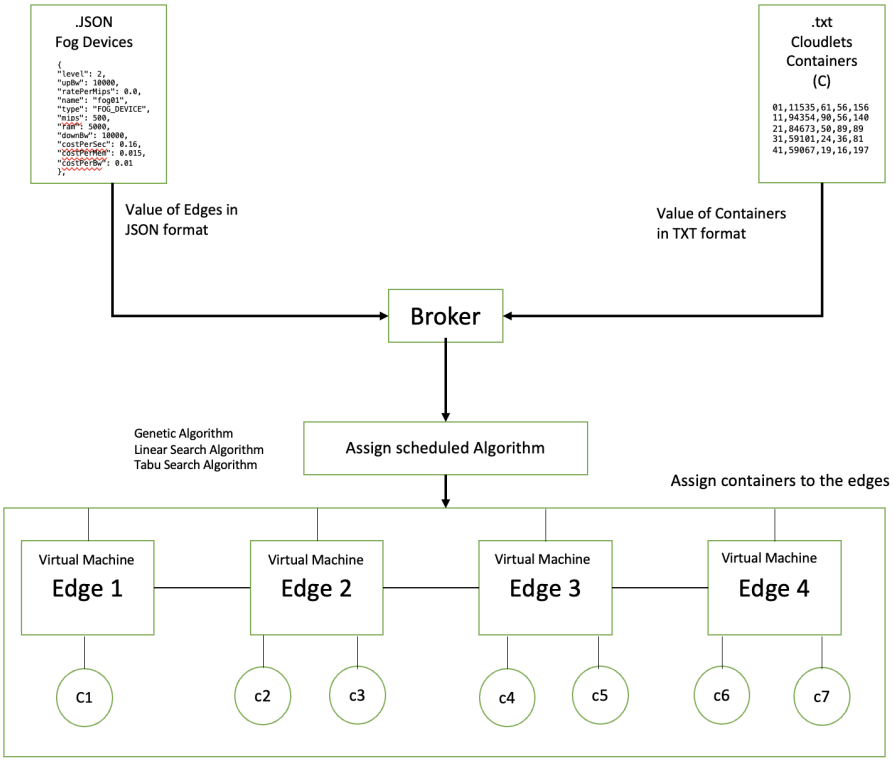


Figure 1: iFogSim Environment Setup

Figure 1 shows two configuration files, the first one is a topology file which is in JSON format and the second one is a container file which is in text format. These files contains

different characteristics of the edges and containers. The characteristics of fog devices are as follows

- MIPS: Million Instructions per seconds is a way to measure processing power. MIPS indicate the processing power of the fog device
- Memory (RAM): It is the amount of Random Access Memory a fog device has.
- Storage capacity: The amount of storage spaces to hold data by a fog device.
- Energy Consumption: Fog devices have different energy consumption profiles.
- Communication Bandwidth: Fog devices have varying communication bandwidths.
- Latency: This is an important characteristics that define the latency that the fog device offers.

Cloudlets also offer the same characteristics as the fog device but it has some unique features like

- Task Offloading: Offloading tasks from end devices, allowing resource intensive task to process locally.
- Virtualization Support: Many cloudlets support virtualization like containers.
- Load Balancing: Cloudlets help in balancing load by redistributing tasks among different fog nodes.

In iFogSim, the FogBroker class is essential for effectively handling scheduling jobs and allocation over the fog network. It acts as a dynamic orchestrator, improving the fog computing environment effectively. FogBroker acts as a mediator between application and fog devices. It also monitors the execution and also manages the resources for fog devices. FogBroker assigns different The characteristics of fog broker class are as follows:

- Task Distribution: The fog broker class uses algorithms to distribute task among different fog devices.
- Scheduling Strategies: Users can implement different scheduling algorithms like metaheuristic algorithms within the FogBroker class.
- QoS Considerations: FogBroker makes sure that tasks with p=higher priority or with strict latency receive timely processing and resource allocation.
- Energy Efficiency: The FogBroker class implements energy efficient scheduling and optimizing task to minimize energy consumption.
- Load Balancing: FogBroker balances the load among different fog nodes.

3.2 Topology Aware Scheduling

The topology that is implemented is shown in the Figure 2. It depicts the topology which has one cloud nodes and with many edges. We want to parameterized the number of nodes to provide topologies with low, medium and large number of edge nodes. Cloud nodes refers to a traditional cloud data center which is located in a centralized location. They are design to process resource intensive task and provide service to wide range of users or edges.

An edge node refers to a device or node that is situated closer to the data source or the end device i.e., in different geographical locations. Edge nodes are crucial for offloading work from centralised cloud nodes, cutting latency, and enhancing the application's overall responsiveness.

Figure 2 shows the design of the system that will be evaluated in later sections. C1, C2 to CN depicts the containers that will be scheduled in these edge devices. Edges 1 to Edges N depict the different number of edges that will be configured. Red lines indicate the connection between the cloud and the edges as well as the edges and the containers. Different latencies are provided between these devices. This topology is modeled for evaluating the scheduling of different meta-heuristic algorithms. The model is scaled and the performance of these algorithms are measured.

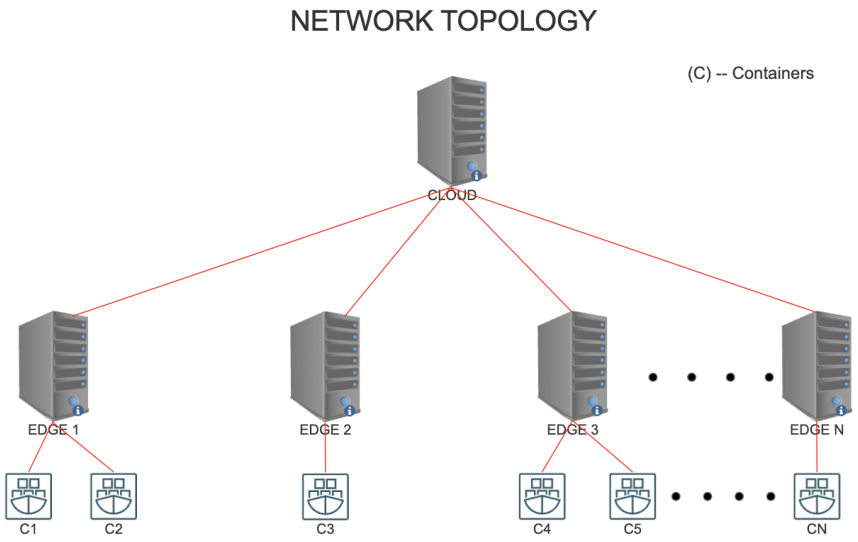


Figure 2: Network Topology

3.3 Metaheuristic Algorithms

Metaheuristic algorithms are at the heart of optimization that can efficiently navigate through solution spaces to find nearly optimal solutions. They have applications, in fields such as optimization, scheduling and routing as they are not limited to problem structures like traditional NP-complete techniques. These algorithms mimic processes like evolution, swarm behavior and other intelligent systems. Examples include colony

optimization, simulated annealing, genetic algorithms and particle swarm optimization. Metaheuristic algorithms effectively tackle problems by balancing exploration and exploitation making them particularly useful in ambiguous or computationally challenging situations. Metaheuristic algorithms support multi-objective optimization and as metaheuristic algorithms provide results that are optimal and not exact, it is good for large scale operations. Minimizing latency and maximizing utilization are two different objectives that can be handled by using metaheuristic approaches. In this research, we are going to compare the following three metaheuristic algorithms.

3.3.1 Genetic Algorithm

In their research paper, Mirjalili and Mirjalili (2019) states that the survival of the fittest principle in nature serves as the basis for Genetic Algorithms (GA). Similar to a population-based game, each solution represents a character with unique qualities (genes). They undergo a fitness test to gauge their performance. The strongest characters have a better probability of getting chosen, but weaker characters nevertheless have a chance. This helps prevent hitting local dead ends. In order to generate new generations, the algorithm combines and modifies the characters, gradually improving the answers. In the end, GA selects the character that fits the situation the best and provides the greatest solution. Algorithm 1 is a pictorial representation taken from a paper published by Katoch et al. (2021). Full implementation is mentioned in the next section4.

Algorithm 1: Genetic Algorithm

Data: Population Y, Population size n, Crossover probability C_p, Mutation probability M_p
Result: New population Y_{new}

```
Initialize population Y with n randomly generated chromosomes;
while Y_new is not complete do
    Select chromosomes C_1 and C_2 from population Y based on their fitness values;
    if Random number r ≤ C_p then
        Apply single-point crossover between C_1 and C_2 to produce offspring O;
    end
    else
        Randomly select one of C_1 or C_2 as the offspring O;
    end
    if Random number r ≤ M_p then
        Apply uniform mutation to offspring O to generate O;
    end
    Place offspring O in the new population Y_new;
end
```

- Initialization: A population (Y) of n chromosomes is randomly initialized.
- The fitness of each chromosome in Y is computed.
- Two chromosomes (C1 and C2) are selected from the population Y based on their fitness values.

- The single-point crossover operator with crossover probability (Cp) is applied to C1 and C2 to produce an offspring (O).
- The uniform mutation operator is applied to the produced offspring (O)with mutation probability(Mp) to generate O.
- The new offspring O is placed in the new population.
- The selection, crossover, and mutation operations are repeated on the current population until the new population is complete.

3.3.2 Local Search Algorithm

Neighbourhood search, commonly referred to as a local search algorithm, is an effective optimisation method for handling difficult issues. It starts with an initial solution and then, by making minor tweaks, investigates surrounding solutions. A fitness function is used to assess the quality of each prospective solution. The algorithm continues to advance towards the best neighbouring solution that increases fitness value, progressively approaching an ideal solution within a constrained region. Neighbourhood search is effective for large-scale issues since it concentrates on locally modifying solutions as opposed to global search techniques.(Ahuja et al. (2002))

Algorithm 2: Local Search Algorithm

Data: Initial solution or state S_{init}, Neighborhood generation function *GenerateNeighborhood*, Fitness function *Evaluate*, Stopping criteria
Result: Optimal solution or local minimum S_{opt}

```
Initialize current solution S_current with S_init;
while Stopping criteria not met do
    Generate neighboring solutions N by applying
     GenerateNeighborhood(S_current);
    Evaluate fitness of each solution in N using Evaluate;
    Select the best neighboring solution S_next that improves fitness;
    if S_next is better than S_current then
        Update S_current to S_next;
    end
    else
        Terminate the loop;
    end
end
return Optimal solution or local minimum S_opt = S_current;
```

- Initialization: Start with an initial solution or state.
- Neighborhood Generation: Generate a solution and generate new neighboring solutions by making small changes or perturbations to the current solution.
- Evaluate: Evaluate the quality of each neighboring solution using a fitness function or an objective function.

- Select best neighbouring solution that improves the fitness value.
- Update Solution : Move to the selected best neighboring solution and update the current solution.
- Termination : Define a stopping criteria and then terminate when the criteria is met or loop-back.

3.3.3 Tabu Search Algorithm

Tabu Search (TS) is a smart search algorithm that remembers its previous moves and uses that knowledge to find better solutions for specific problems. It keeps track of historical data in its memory, which helps it make well-informed decisions during the search process. The memory is divided into two types: explicit memory, which stores complete and highly attractive solutions to explore neighborhoods effectively, and attributed memory, which stores data based on important features to create optimal solutions. Both short-term and long-term memory structures, each with specific characteristics and methods, are used by TS. It adheres to the problem's requirements while making advantage of clever move methods like switching chosen edges. With its memory and cunning manoeuvres, TS develops into an effective tool that strengthens search tactics and overall performance.Algorithm 3 is taken from (Prajapati et al. (2020)).

Algorithm 3: Tabu Search Algorithm

Data: Initial solution or state S_{init}, Neighborhood generation function *GenerateNeighborhood*, Fitness function *Evaluate*, Tabu list size T, Number of iterations N

Result: Best solution found S_{best}

```
Initialize current solution S_current with S_init;
Initialize Tabu list T_list as an empty list;
Initialize best solution S_best with S_current;
for i ← 1 to N do
    Generate neighboring solutions N by applying
     GenerateNeighborhood(S_current);
    Evaluate fitness of each solution in N using Evaluate;
    Select the best non-tabu solution S_next from N;
    Add S_next to the Tabu list T_list;
    if Size of T_list exceeds T then
        Remove oldest entry from T_list;
    end
    if S_next is better than S_best then
        Update S_best to S_next;
    end
    Update S_current to S_next;
end
return Best solution found S_best;
```

- Initialization: initializing an initial solution and then a restricted Tabu list for tracking. Along with that initializing a variable to retain the most favorable solution.

- Iterative Exploration: the algorithm generates neighboring solutions through a neighborhood generation function, assesses their fitness using a suitable fitness function, and selects the most favorable non-tabu neighboring solution.
- Tabu List Management: The chosen solution is put into the Tabu list and is marked as forbidden for certain iterations.
- Solution Update: If the selected solution better than the current best solution, it replaces the latter and again starts the cycle of exploration and exploitation.

3.3.4 Discussion on Algorithms

The above algorithms in considerations are very popular and have unique features and advantages. The strength of **Genetic Algorithm** lies in its ability to explore search spaces simultaneously. Exploration means the algorithms ability to search new areas where it could find a possible solution. Genetic Algorithm can search multiple potential areas parallel which means that it can execute and schedule containers faster. On the other hand **Local Search Algorithm** has the ability to fine tune its algorithm which means that it does not need to explore the whole search space. It means that this algorithm can benefit by saving costs. **Tabu Search Algorithm** maintains a adaptive Tabu memory which in theory help keep record of the search spaces already visited and it prohibits the re-visitation of the sub optimal solution. Thus in the evaluation section, we will look into the different evaluation metrics and compare the performance of these algorithms on the basis of execution time and cost.

In the next section, we will look into the design specification and implementation of these algorithms.

4 Design Specification & Implementation

4.1 Topology Architecture & Specifications

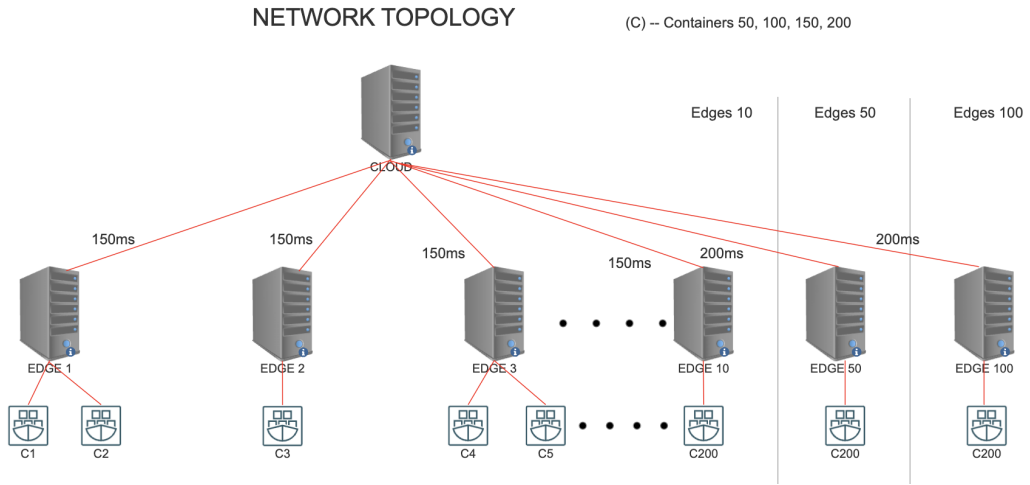


Figure 3: Topology Architecture

The topology architecture in Figure 3 is scaled starting from 10 edges, 50 edges and 100 edges. We schedule 50, 100, 150 and 200 containers at every edge interval. The network latency between the cloud and the edge node vary from 150ms to 200ms which is taken from a conference. [1]

4.1.1 Node Configuration

This architecture contains 10 edges which gradually increases to 50 and 100 and with 1 cloud node. The following Figure 4 shows the configuration of the cloud nodes and the edge nodes. This configuration is in the .JSON format and is executed by the fog broker entity of iFogSim. All the configurations are taken from paper written by Awaisi et al. (2019).

Parameters	Edge Nodes	Cloud Nodes
CPU Rate (MIPS)	[500, 1500]	[3000, 5000]
CPU Uage Cost	[0.1, 0.4]	[0.7, 1.0]
Memory Usage Cost	[0.01, 0.03]	[0.02, 0.05]
Bandwidth Usage Cost	[0.01, 0.02]	[0.05, 0.1]

Figure 4: Node Configuration

4.1.2 Container Configuration

The .txt file contains the details of the containers that is going to be scheduled by different algorithms. All the configurations are taken from [2] The configuration of the containers are given in the following Figure 5

Property	Value
No of Instructions(10^9)	1
Memory Required(MB)	[50, 200]
Input File Size (MB)	[10, 100]
Output File Size (MB)	[10, 100]

Figure 5: Container Configuration

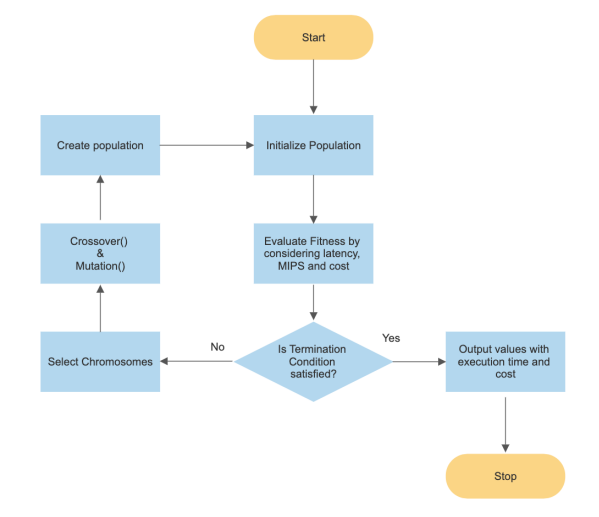
[1] https://www.thousandeyes.com/resources/cloud-performance-report-2022
[2] https://www.kaggle.com/datasets/sachin26240/vehicularfogcomputing

4.2 Algorithms

There are three different algorithms that are being tested in this paper namely Genetic Algorithm, Local Search Algorithm and Tabu Search Algorithm. The design and implementation of these algorithms are as follows

4.2.1 Genetic Algorithm

The configuration of genetic algorithm is shown in the Figure 6. Number of individuals is defined as the size of the population or can also be called as collection of potential solution. The value is set to 500 which is optimal for this problem. Number of iteration is also known as the number of generation that GA will run the evolutionary process. More iteration leads to better convergence but it can also lead to an increase in cost. And the value is set to 1000. Mutation rate brings in random change to an individual. Increasing mutation rates promotes exploration of the search space but it gives poor convergence, thus the value is set to 0.01. Crossover rate is mainly used to exploit the already given search space and the value is set to 0.8. Number of Elicit individual is defined as the best individual from the current generation and the value is set to 1.

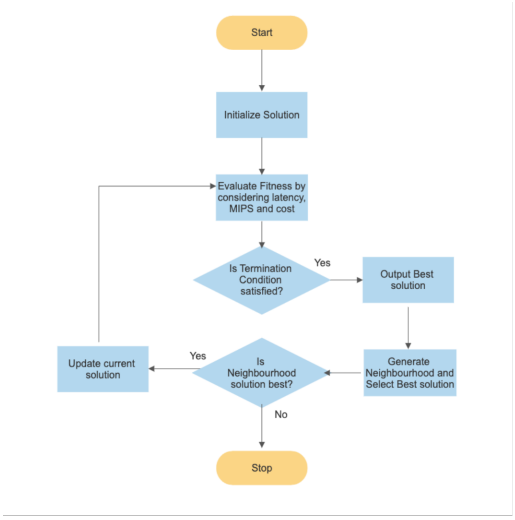


GA Parameter Setting	Value
NUMBER OF INDIVIDUAL	500
NUMBER OF ITERATION	1000
MUTATION_RATE	0.01
CROSSOVER_RATE	0.8
NUMBER_ELITISM_INDIVIDUAL	1

Figure 6: Genetic Algorithm Implementation

4.2.2 Linear Search Algorithm

The configuration of Linear Search Algorithm is shown in Figure 7. An iterative improvement strategy is used in local search to investigate the area around a particular solution. This variable is known as local search iteration and the value is set to 200. The term initial sacrifice can be referred to as perturbation. It is basically given to introduce randomness or diversification in the local search process. The method seeks to explore a larger search area that could include better answers by first compromising the quality of the present result and the value is set to 0.005. The descending speed in local search algorithms often refers to the step size or movement distance covered during each iteration of neighbouring solution exploration. The algorithm decides how far to step or modify the solution while travelling from the present solution to its neighbours and the value is set to 1.1.

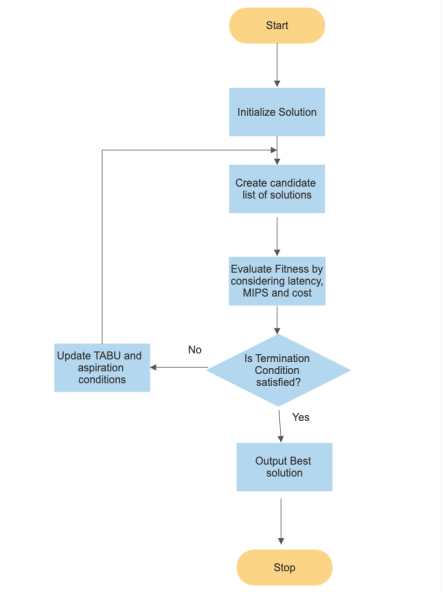


Local Search Parameter Setting	Value
LOCALSEARCH_ITERATIONS	200
INITIAL_SACRIFICE	0.005
DESCENDING_SPEED	1.1

Figure 7: Linear Search Algorithm Implementation

4.2.3 Tabu Search Algorithm

The configuration of Tabu Search Algorithm is shown in the Figure 8. MaxStable parameter is also known as the maximum stagnation or the maximum value of non improving iterations to the best solution. Stagnation occurs when the algorithm remains trap in the local optima. Thus the max value is set to 100. MaxIteration is also defined as the maximum value of iteration where the algorithm will run and it is 1400. Maximum time is the max time the algorithm will run and is set to 20. TabuLength is an important and it is defined as how long a solution is considered as tabu or forbidden. When a move is performed, it is added to a tabu list and no changes are made till the tabu length. The value of tabu is set to 30.



Tabu Search Parameter Setting	Value
maxStable	100
maxInteration	1400
maxTime	20
tabulength	30

Figure 8: Tabu Search Algorithm Implementation

4.3 KubeEdge

An open source technology known as KubeEdge allows hosts at the Edge to enhance native containerized application orchestration capabilities. It is based on Kubernetes and offers basic network, application deployment, and metadata synchronisation between cloud and edge infrastructure support. KubeEdge is preferred because the master control plane is in the cloud and the workers are in edges. It helps in control plane separation. After the implementation of algorithms, a YAML file can be generated which in turn can be implemented in the KubeEdge to schedule containers.

5 Evaluation

The evaluation of results conducted will be explained in this section. The experiments are conducted in iFogSim simulator with simulation of one topology and scaling the topology with different edge nodes. Along with that a comparative analysis of response time and cost of three algorithms have been conducted. With each experimental iteration, the quantity of containers increases, ranging from 50 to 500, The simulation is divided into three experiments each providing the graphs for execution time and cost.

- Experiment 1: Consists of 10 edge nodes with 50, 100, 150, 200, 300, 500 containers.
- Experiment 2: Consists of 50 edge nodes with 50, 100, 150, 200, 300, 500 containers
- Experiment 3: Consists of 100 edge nodes with 50, 100, 150, 200, 300, 500 containers

5.1 Experiment 1 : 10 Edges

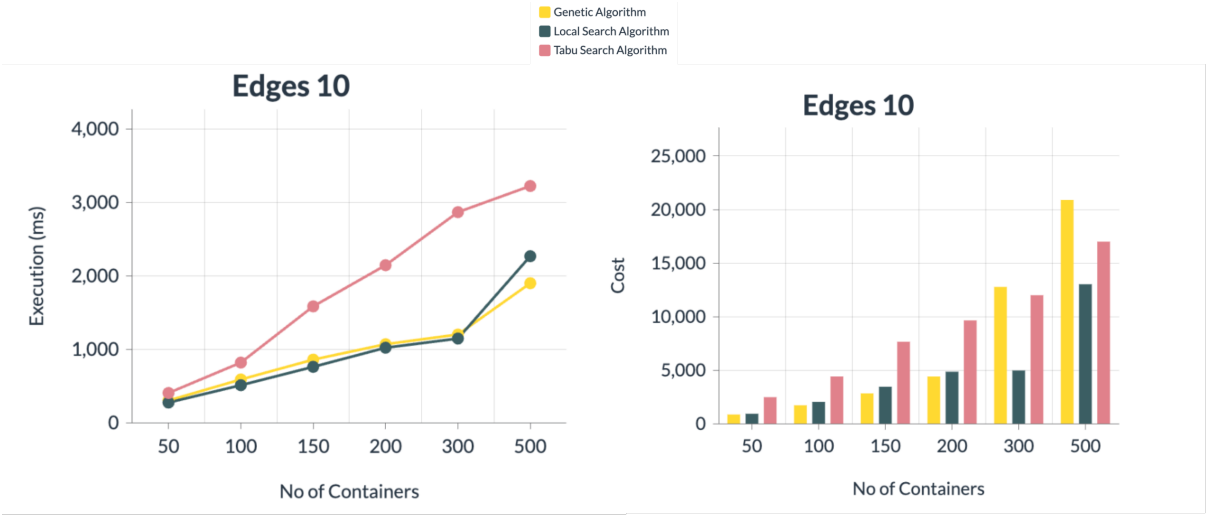


Figure 9: Experiment 1: 10 Edge nodes

The first experiment is carried out with 10 Edge nodes and spinning up containers ranging from 50 to 500 and noting down the response time (Execution time) as well as the cost to execute these containers. From Figure 9 it can be concluded that Genetic Algorithm and Local Search algorithm performs in a similar way but when scaling up, Genetic Algorithm outperforms Local Search algorithm. The major difference is seen when scheduling 500 containers where execution time of GA is less that Local Search. Execution time of Tabu Search algorithm is significantly higher than the other two. And comparing costs, it can be concluded that when scaling up the number of containers, the cost of Local Search algorithm decreases as compared to Genetic Algorithm which has the highest peak.

5.2 Experiment 2 : 50 Edges

The second experiment is carried out by scaling up the Edges to 50. From the Figure 10, the difference in execution time between GA and LSA increases, which shows that GA performs well in complex large scale scenarios. When compare costs, it is seen that LSA algorithm is more cost effective than GA and Tabu search as well.

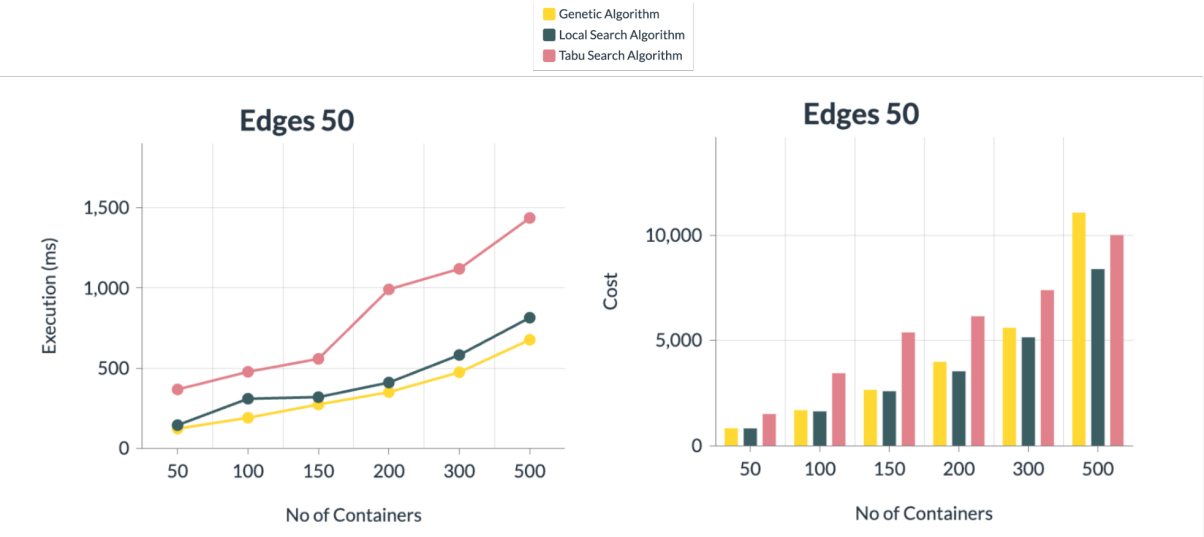


Figure 10: Experiment 2: 50 Edge nodes

5.3 Experiment 3 : 100 Edges

The third experiment is carried out by scaling up the edges to 100. From the Figure 11, the difference in execution time further increases when comparing GA to LSA, where GA outperforms LSA. And comparing costs, it can be seen that LSA is slightly more cost effective in large scale applications.

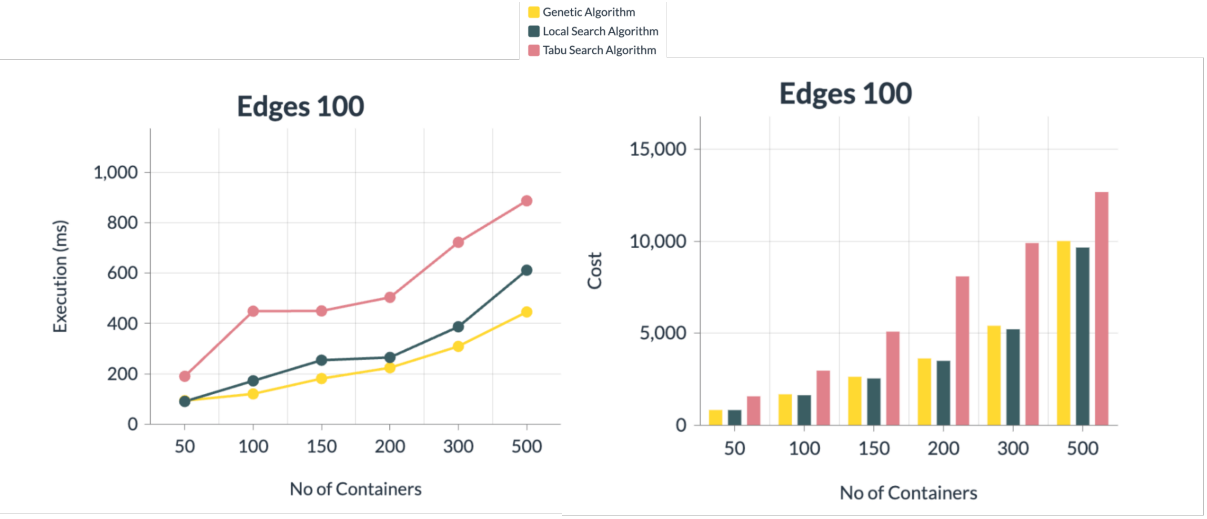


Figure 11: Experiment 3: 100 Edge nodes

5.4 Discussion

In this section, lets deep dive into the evaluation of different metrics and the results of the above graphs.

- From Figure 9, Local Search Algorithm (LSA) maintains the lowest execution time upto 300 containers. And when scaling up above 300, Genetic Algorithm (GA) performs better. GA has 16.24% less latency than LSA when scaled up to 500 containers and that gap keeps increasing. And when comparing costs, it can be seen that GA performs slightly better than LSA when scaling upto 200 containers. When scaling up above 200, the cost spikes up for GA and LSA maintains a steady increase.

- Tabu Search Algorithm (TSA) performs bests with low number of edges and containers. TSA exhibits similar performance as the other two algorithms for 10 edges and upto 100 containers. Hence, it is not recommended for scheduling large scale application.

- From Figure 10, when scaling up the edge node to 50, the execution time of GA is less than LSA and the percentage gap increases to 16.98%. Thus GA is recommended over LSA for 50 fog nodes and when comparing costs, LSA algorithm outshines other two algorithm with difference in cost of approximately 20%. When scheduling 500 containers, Tabu Search Algorithm (TSA) performs better than GA.

- From Figure 11, when scaling from 50 to 100 edges, the gap between execution time increases from 16.24% to 18% between GA and LSA, GA being with a better execution time. When scheduling 200 containers,a similar execution time is noticed between GA and LSA. From the above experiments, Genetic Algorithm is recommended for scheduling large scale applications having execution time constrain. Now comparing costs, Local Search Algorithm performs better with having approximately 9% decrease in cost than Genetic Algorithm for scheduling large scale applications.

6 Conclusion and Future Work

In this paper, a comparison between three different meta-heuristic algorithms namely Genetic Algorithm, Local Search Algorithm and Tabu Search Algorithm have been performed successfully. The recommended Genetic Algorithm(GA) performs better than the other two algorithm in terms of execution time. Local Search Algorithm(LSA) has a similar performance but when scaling up the edges, GA outperforms LSA. Local Search Algorithm performs better than other two algorithms in terms of optimizing costs. This research successfully demonstrates the comparison between three different algorithm in topology-aware scenarios. These experiments were conducted on a simulation tool called iFogSim due to time and cost constrains, the future works include the following:

- Creating a YAML file with output of these algorithms.
- The implementation of these algorithm on actual Kubernetes cluster built for fog and edge scenarios using KubeEdge.
- Another work will be to analyze different algorithms in network-aware topology where the pods are interconnected and dependent on each other.

References

Ahuja, R. K., Özlem Ergun, Orlin, J. B. and Punnen, A. P. (2002). A survey of very large-scale neighborhood search techniques, *Discrete Applied Mathematics* **123**(1): 75–102.
URL: *https://www.sciencedirect.com/science/article/pii/S0166218X01003389*

Awaisi, K. S., Abbas, A., Zareei, M., Khattak, H. A., Shahid Khan, M. U., Ali, M., Ud Din, I. and Shah, S. (2019). Towards a fog enabled efficient car parking architecture, *IEEE Access* **7**: 159100–159111.

Calheiros, R. N., Ranjan, R., Beloglazov, A., De Rose, C. A. and Buyya, R. (2011). Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms, *Software: Practice and experience* **41**(1): 23–50.

Caminero, A. C. and Muñoz-Mansilla, R. (2021). Quality of service provision in fog computing: Network-aware scheduling of containers, *Sensors* **21**(12): 3978.

Eidenbenz, R., Pignolet, Y.-A. and Ryser, A. (2020). Latency-aware industrial fog application orchestration with kubernetes, *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, pp. 164–171.

Fu, K., Zhang, W., Chen, Q., Zeng, D., Peng, X., Zheng, W. and Guo, M. (2021). Qos-aware and resource efficient microservice deployment in cloud-edge continuum, *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, pp. 932–941.

Haja, D., Szalay, M., Sonkoly, B., Pongracz, G. and Toka, L. (2019). Sharpening kubernetes for the edge, *Proceedings of the ACM SIGCOMM 2019 Conference Posters and Demos*, pp. 136–137.

Katoch, S., Chauhan, S. S. and Kumar, V. (2021). A review on genetic algorithm: past, present, and future, *Multimedia tools and applications* **80**: 8091–8126.

Kayal, P. (2020). Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper, *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, pp. 1–6.

Marchese, A. and Tomarchio, O. (2022). Network-aware container placement in cloud-edge kubernetes clusters, *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pp. 859–865.

Mirjalili, S. and Mirjalili, S. (2019). Genetic algorithm, *Evolutionary Algorithms and Neural Networks: Theory and Applications* pp. 43–55.

Nakanishi, K., Suzuki, F., Ohzahata, S., Yamamoto, R. and Kato, T. (2020). A container-based content delivery method for edge cloud over wide area network, *2020 International Conference on Information Networking (ICOIN)*, pp. 568–573.

Nguyen, N. D., Phan, L.-A., Park, D.-H., Kim, S. and Kim, T. (2020). Elasticfog: Elastic resource provisioning in container-based fog computing, *IEEE Access* **8**: 183879–183890.

Perez Abreu, D., Velasquez, K., Curado, M. and Monteiro, E. (2019). A comparative analysis of simulators for the cloud to fog continuum, *Simulation Modelling Practice and Theory* **101**: 102029.

Prajapati, V. K., Jain, M. and Chouhan, L. (2020). Tabu search algorithm (tsa): A comprehensive survey, *2020 3rd International Conference on Emerging Technologies in Computer Engineering: Machine Learning and Internet of Things (ICETCE)*, pp. 1–8.

Pusztai, T., Rossi, F. and Dustdar, S. (2021). Pogonip: Scheduling asynchronous applications on the edge, *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, IEEE, pp. 660–670.

Rejiba, Z. and Chamanara, J. (2022). Custom scheduling in kubernetes: A survey on common problems and solution approaches, *ACM Comput. Surv.* **55**(7).
URL: *https://doi.org/10.1145/3544788*

Rossi, F., Cardellini, V., Presti, F. L. and Nardelli, M. (2020). Geo-distributed efficient deployment of containers with kubernetes, *Computer Communications* **159**: 161–174.

Santos, J., Wauters, T., Volckaert, B. and De Turck, F. (2019). Towards network-aware resource provisioning in kubernetes for fog computing applications, *2019 IEEE Conference on Network Softwarization (NetSoft)*, IEEE, pp. 351–359.