FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Multi-objective User and Virtual Machine Assignment Using Biogeography-Based Optimization

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Software Engineering and Internet Computing

eingereicht von

Florian Rohrer, BSc.
Matrikelnummer 0826568

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Ivona Brandic

Wien, 22. Dezember 2017

Florian Rohrer	Ivona Brandic

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Multi-objective User and Virtual Machine Assignment Using Biogeography-Based Optimization

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Software Engineering and Internet Computing

by

Florian Rohrer, BSc.
Registration Number 0826568

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Ivona Brandic

Vienna, 22nd December, 2017

Florian Rohrer	Ivona Brandic

Erklärung zur Verfassung der Arbeit

Florian Rohrer, BSc.
Karlsplatz 13, 1040 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 22. Dezember 2017

Florian Rohrer

Danksagung

Ich möchte meiner Betreuerin Dr. Ivona Brandic von der Technischen Universität Wien für ihre kontinuierliche Unterstützung danken. Die fachlichen Diskussionen haben mir wertvolle Einblicke verschafft und waren eine unbezahlbare Ressource für mich. Ich hatte die Chance, mein Projekt nach meinen eigenen Vorstellungen zu gestalten, konnte aber immer die Expertise und Führung meiner Betreuerin einholen.

Ich möchte meinen Freunden danken, die ständig kritisch sind und die nur das Allerbeste akzeptieren.

Schließlich möchte ich meinen Eltern danken, die mich während der gesamten Studienzeit ständig unterstützt haben. Sie hatten durchgehend Geduld und eine unerschütterliche Zuversicht in meine Fähigkeiten. Ich bin dankbar und fühle mich zutiefst geehrt.

Danke.

Acknowledgements

I would like to thank my thesis advisor Dr. Ivona Brandic of the Vienna University of Technology for her continuous support. The technical discussions and valuable insights I gained from her, were an invaluable resource. She allowed this project to be my own, yet helped me find to my way, when I needed expertise and guidance.

I would like to thank my friends for constantly challenging me to do my best and never excepting anything less than excellence.

Finally, I would like to thank my parents, who provided me with support throughout all those years of study. They were tremendously patient and always had faith in my abilities. I am grateful and honored.

Thank you.

Kurzfassung

Entwicklungen wie intelligente Verkehrssysteme, innovative Stromnetzüberwachung und -regelung sowie Echtzeitspiele, die mit der Umgebung interagieren, sind neuartige Applikationen, die einen hohen Anspruch auf geringe Latenzzeit stellen. *Fog Computing* ist ein neues Programmierparadigma, das den Weg für solche Applikationen ebnen wird. Dazu werden Serviceanbieter ihre Hardware (Gateways, Router, Server) außerhalb von Rechenzentren positionieren, sodass sie sich physisch so nahe wie möglich bei ihren Endnutzern befindet. Die Nutzer können sich allerdings frei bewegen und müssen daher abhängig von ihrem Standort, den Servern zugewiesen werden. Gleichzeitig müssen die Applikationen der Nutzer ebenfalls diesen Servern zugewiesen werden, idealerweise so, dass sie sich möglichst nahe zu ihnen befinden. Unterschiedliche Ziele von Serviceanbietern und Nutzern sowie physische Limitierungen der Hardware, machen dieses Zuweisungsproblem zu einer herausfordernden Aufgabe. Wir erfassen die Ziele von Nutzern und Serviceanbietern in sechs verschiedenen Zielfunktionen (*user distance*, *power consumption*, *resource waste*, *failure probability*, *reachability*, *user evenness*). Wir verwenden *Biogeography-based optimization* (BBO) eine spezielle Art eines genetischen Algorithmus inspiriert von einem Prozess aus der Natur, um Lösungen zu finden, die gleichzeitig alle dieser Zielfunktionen minimieren. Da es sehr schwierig ist, exakte Lösungen zu finden, vergleichen wir BBO mit einem Greedy-Algorithmus und einem Genetischen Algorithmus (GA). Unsere ausgiebigen Simulationen zeigen, dass BBO tatsächlich in der Lage ist, akzeptable Lösungen zu finden, allerdings abhängig von der beobachteten Zielfunktion. BBO liefert bessere Lösungen als der Greedy-Algorithmus in fast allen Test-Instanzen und geringfügig bessere Lösungen als GA, besonders in kleineren Instanzen. Diese Arbeit zeigt, dass BBO ein geeignetes Verfahren ist, um dieses Zuweisungsproblem, das in Fog Computing auftritt, zu lösen und kann zu kleineren Latenzzeiten für Endnutzer einerseits und geringeren Kosten und höheren Verfügbarkeiten für Serviceanbieter andererseits führen.

Abstract

Applications such as vehicle routing systems, smart city applications and augmented reality games are new and demanding applications that have very low latency requirements. *Fog Computing* is a new computing paradigm that will pave the way for these applications. To that end, service providers will deploy their hardware (gateways, routers, servers) outside of data centers in order to move their infrastructure as close as possible to their users. In order to get benefit from the infrastructure, users have to be assigned to the right servers, depending on their physical location. Simultaneously, the users' applications, have to be assigned to the same servers, ideally such that they are in close proximity to their users. The different objectives of both service providers and users, as well as physical limitations and constraints, however, make this assignment task a challenging problem. We capture user and service provider goals by six different objective functions (*user distance*, *power consumption*, *resource waste*, *failure probability*, *reachability*, *user evenness*). We use *Biogeography-based optimization* (BBO), a kind of genetic algorithm inspired by nature, to find solutions that simultaneously minimize all these functions in a multiobjective Pareto approach. As exact solutions are hard to obtain, we compare BBO against a greedy algorithm and the Genetic algorithm (GA). Our extensive simulations suggest that BBO is indeed applicable to find reasonably good solutions, however results vary upon the used objective function. BBO generally outperforms the greedy algorithm in almost all of the instances and often delivers slightly better results than GA as well, especially for smaller instance sizes. The work shows that BBO is an admissible approach for solving this assignment problem encountered in Fog Computing which can lead to lower latency for user applications and cost savings and higher availability guarantees for service providers.

Contents

CHAPTER 1

Introduction

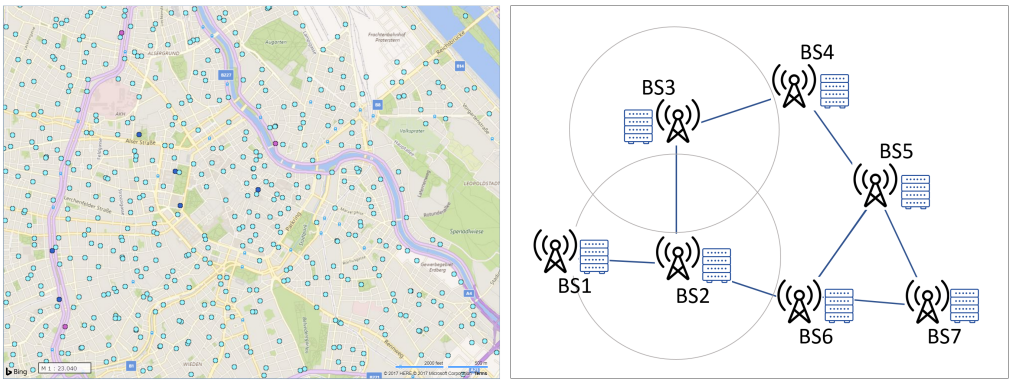
1.1 Problem Definition

More and more physical devices are connected to the Internet. According to a recent report by Gartner, an estimated 8.38 billion devices will be online by the end of 2017 and this number is expected to grow to 20.4 billion by the end of 2020 [4]. Examples range from fixed *things* such as household devices, electrical grid sensors and surveillance cameras to mobile things such as cars, smartphones and wearables like watches and fitness trackers. In addition to the network, they can also be connected to actuators, which allows them to not only track and control the environment, but also manipulation of it for regulation purposes.

These *smart* devices in their entirety constitute what is now called the *Internet of Things* (IoT). The benefits of IoT are manifold. One of the biggest benefits is expected in supply chain management and asset tracking. Logistics company DHL stated in a press release in 2015 that it expects a USD 1.9 trillion boost to their supply chain and logistics operations, due to improved decision-making in warehouse operations and near real-time data analytics based on smart devices [10]. Another huge opportunity for IoT is efficient energy management: Energy use within residential homes (lighting, cooling, heating systems) and energy use for transportation (personal vehicles and public transportation) can be cut down through IoT-enabled solutions, through intelligent operation of activities on one hand and traffic management, congestion control, and smart parking, on the other [31].

IoT devices are said to be at the *edge* of the network, because they are at the boundary to the user. Since the number of edge devices grows rapidly, we are now facing a critical issue: The amount of data produced will soon reach and then surpass the amount of data that the network is capable of transmitting to cloud data centers for processing that are situated further away. A possible remedy is processing the data earlier, immediately

after they were recorded. This requires a new architecture, that includes *edge data centers* (EDCs). These small, light-weight distributed machines form an intermediate network and are capable of storing, aggregating and processing data in near real-time at the edge. Figure 1.1 shows how such an architecture would look like.



(a) **(b)**

Figure 1.1: (a) Base stations in and around the center of Vienna, Austria [16]. Light blue dots indicate cell towers, magenta dots indicate radio towers and dark blue dots indicate auxiliary measurement stations. (b) Envisioned derived fog architecture with Edge Data Centers (EDCs) attached to base stations [36].

Figure 1.1 (a) shows an example of how many base stations exist in a densely populated area. Figure 1.1 (b) depicts the imagined servers or physical machines (PMs) that would be co-located with base stations, to enable the aforementioned processing capabilities. We will refer to this architecture as *fog* or *Fog Computing* and will use it as the basis for our study[1].

IoT devices (such as smartphones) are often resource-limited. Since EDCs provide processing capabilities, these limitations can be alleviated in order to enhance mobile user experience: *Computation offloading* is the practice of offloading expensive computation tasks to another entity, with the goal of extending battery life and/or increasing computational capabilities of the first entity [50]. The mobile assistant tool *Siri* [3] is a good example of how voice recognition can be implemented by computation offloading [7]. Leveraging the close proximity of EDCs, even more demanding applications will be possible in the future that require more network bandwidth or shorter response times [87].

Virtual Machines (VMs) have been around for many years now and their popularity is due to two key benefits. First, they are isolated execution environments capable of running user tasks and applications. This is, they provide all the capabilities of an operating system. And second, they are location independent, meaning that they can be assigned to and run on any PM. For example, in cloud data centers with many PMs, it is common practice to assign VMs to PMs in order to distribute computation load evenly. We will

[1]Clouds consist of vapor that is *far away* from the ground where all the humans (users) reside. Fog consists of vapor that is *near* to the ground and the users, hence the term *Fog Computing*.

Table 1.1: Number of possible solutions for the VM/User assignment problem.

# PMs	# VMs	# Users	$\lvert\Sigma\rvert$
25	25	25	$7.52 \cdot 10^{46}$
50	50	50	$6.37 \cdot 10^{108}$
75	75	75	$2.59 \cdot 10^{176}$
100	100	100	$5.15 \cdot 10^{247}$

view VMs as containers for user applications (apps), thus making the user applications location independent.

Because of these two benefits, isolation and location independence, VMs are a prime candidate to be used in a fog computing environment. The idea is that users connect to a EDC in their vicinity and the applications they need (encapsulated in the form of VMs), are subsequently transferred to the same EDC. If more than one EDC is in a user's vicinity, one has to be selected. EDCs, however, can host only a certain number of VMs due to physical capacity constraints. If a user needs a VM that cannot be hosted on the EDC they are currently connected to, the VM can also be hosted on a different EDC. In that case, the user request will then be routed to this different EDC. This incurs some delay. How to assign both users and VMs to a set of edge data centers such that delay and other costs are minimized, will be explored in this thesis.

Due to the combinatorial nature of this problem, the number of possible solutions is very large, even if the numbers of EDCs, VMs and users is relatively small. Let Σ be the set of all possible solutions to the problem. Assume, any VM can be assigned to any EDC (ignoring physical limitations for the moment) and that on average three base stations are in the vicinity of a user. Then the size of this set is given by

$$|\Sigma| = M^N \cdot 3^U, \tag{1.1}$$

where M, N and U are the number of PMs, VMs and users, respectively. An illustrative example of how fast this number grows, is given in Table 1.1.

We will assume the network that connects all PMs is a virtualized, *bandwidth guaranteed* network and assume that enough bandwidth is available for both user requests from a base station to an EDC and from one EDC to another for relaying and inter-EDC communication purposes. Establishing such bandwidth guarantees on a shared network infrastructure is a complex and challenging task. Yet, there is myriad of applications that successfully tackle this task such as *SecondNet* [38], *ElasticSwitch* [66], *CloudMirror* [53], and very recently *DetServ* [37].

1.2 Motivation and Contribution of the Thesis

Multi-objective Optimization in Fog Computing Environment

In this thesis, we propose a new approach for efficiently assigning both users and VMs to small-sized data centers (EDCs) that are geographically dispersed over a large area.

Gu et al. [36] studied a similar scenario, namely cost efficient resource management for medical systems relying on Fog Computing. Their cost function is a single function which consists of communication costs between clients and base stations, inter-base station communication and VM deployment cost. Further, the researchers used a linear programming (LP) based heuristic for minimizing this cost function. Our work differs in several ways. First, we used a different set of objective functions, that shifts the focus away from only considering communication costs. Our approach is capable of balancing service provider goals on one hand and user requirements on the other. Second, LP is capable of minimizing only a single function and therefore the researchers used the sum of objective functions to derive a single function (returning a scalar value). We use several individual cost functions (a cost vector) that are jointly minimized, using the concept of Pareto optimality. This avoids the use of the summation or some other kind of aggregating function and further allows us to incorporate objective functions that are not comparable with each other because, for example, they are measured on different scales or use different units. Third, we use a computationally less expensive solving technique for finding the minima of these cost functions. This may yield results sooner.

Our approach relies on a special variant of genetic algorithm called *Biogeography-based optimization* (BBO). Genetic algorithms in general have a variety of benefits, that make them an attractive choice for solving assignment problems. One of the biggest benefit is that they impose no restrictions on the nature of the objective functions they are trying to optimize. Genetic algorithms can be used on non-linear functions, functions that are not differentiable (since they do not calculate a gradient) and even on discontinuous functions. This makes them applicable to a wide range of problems. The classical genetic algorithm (GA) (as described for example in [61]) solves optimization problems using a linear solution encoding, followed by a *selection*, *crossover* and *mutation* process of those solutions. BBO was introduced in 2008 by Simon [73] and refined some steps of GA, namely selection and crossover. We chose BBO as the basis of this work for two reasons: First, BBO is more recent and it has been shown that the refinements made by Simon make BBO outperform GA in various scenarios [73, 89, 76]. Second, to the best of our knowledge, BBO has not been used in a Fog Computing environment. It has already been used in a Cloud Computing setting [21, 91] and had better convergence characteristics and was more computationally efficient than the used reference strategies [90], which is a good premise for a successful study in the Fog Computing realm.

Fog Environment Simulation Scripts

Evaluation of this work will be done using a simulation tool specifically written for this purpose using MATLAB. Our workflow consists of two major steps: The first step is the creation of the Fog environment including users and their respective locations as well as VMs. The second step is the subsequent execution of BBO to find an efficient assignment of users and VMs to the fog hardware. In the first step, the tool relies on certain modeling assumptions including the assumed hardware specifications for physical machines (obtained from SPEC.org [17]) and assumed network connection layout (that is, the graph structure between PMs). Most of these parameters of the simulation environment can be altered, making the developed tool customizable to specific Fog computing scenarios. Therefore, service providers could use the tool to find user and VM assignments in their own fog environment. In the second step, execution of BBO, the service provider can also intervene and alter some of the BBO parameters. This gives the service provider the opportunity to instruct the tool to use more time or a different mutation strategy which in turn influences the final solution quality.

The primary contributions of this thesis can be summarized as follows:

(1) We abstract the assignment problem and derive several objective functions in order to study the problem analytically.

(2) We use Biogeography-based optimization, a special variant of genetic algorithm, and make certain adaptations to its original formulation in order to make it applicable for both a discrete assignment problem and a problem with multiple objective functions.

(3) We conduct extensive simulations to evaluate the algorithm parameters influence its efficiency, and demonstrate the efficacy of our approach using various experiment instance sizes.

1.3 Structure of the Thesis

This thesis is structured as follows. In Chapter 2, we discuss the background of the thesis and will introduce the concepts of cloud and fog computing. Also, virtual machines and their migration mechanisms will be discussed. Then the optimization technique used will be explained, including some adaptations that were made to it.

Chapter 3 reviews current approaches for load balancing. Our focus will be on assignment that uses heuristics and metaheuristics are their main mechanisms. Other approaches will be discussed as well. Specific fog environments challenges and solutions will be highlighted.

In Chapter 4, we describe the assignment problem in more detail, define the goals we are trying to achieve and derive from these goals several objective functions, that we will

later minimize. Also in this Chapter, we explain how the optimization technique was used and modified to solve our specific problem at hand.

Chapter 5 contains the evaluation of our approach, including the experiment setup, descriptions of the reference algorithms and results and discussion. Lastly, we conclude give pointers to possible further research directions and conclude was has been learned.

CHAPTER 2

Background

2.1 Fog Computing Environment

Fog Computing is a new computing paradigm, that will be reality in the near future as more and more devices are connected to the Internet. It will complement the Cloud Computing paradigm, that many applications rely on today and will pave the way for a whole new type of more demanding and capable applications in the future. In this Section, first, we look at the definition of Cloud Computing and explain its different flavors, SaaS, PaaS and IaaS. Second, we introduce Fog Computing and the benefits it holds for applications. Finally, we follow-up with a discussion about Virtual Machines and the huge potential they offer.

2.1.1 Cloud Computing

In recent years, *cloud computing* has become a popular term for a computing paradigm that offers certain desirable features. *Cloud providers* provide all the hardware and part of the software stack necessary for storage of data and computation of tasks, while *cloud users* use these resources by implementing their own cloud applications on top of them and/or use existing tools and applications that are offered. The U.S. National Institute of Standards and Technology (NIST) defines cloud computing as follows [59, p. 2]:

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*

Voorsluys et al. argue that commonality among many cloud service providers is a *pay-per-use* model, that involves no ongoing commitment and utility prices and that resources

are provided in an abstracted or *virtualized* way. Further, users get elastic capacity and the impression of having access to infinite resources [82, p. 4].

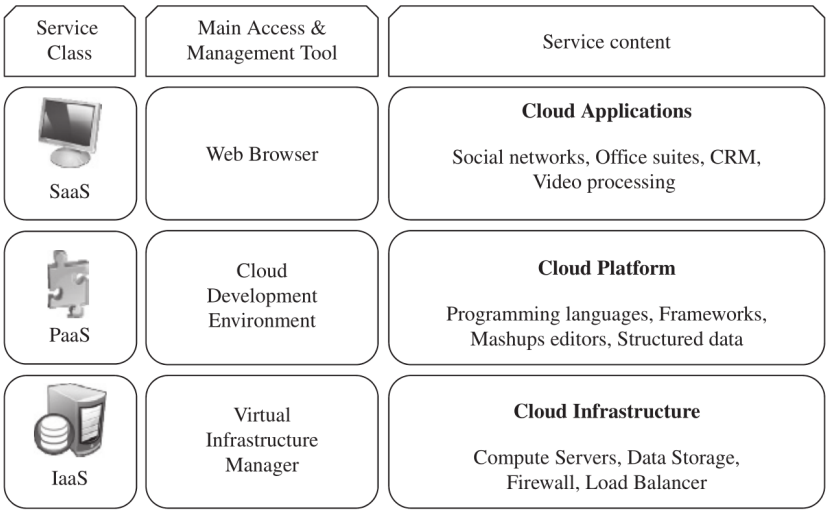


Figure 2.1: Different types of services offered by Cloud Computing provides [82, p. 14]

Many commercial cloud computing services exist, such as Amazon web services [2], Google Cloud Platform [5], Microsoft Azure [12] and IBM Bluemix [9]. These services differ in offered functionality as well as in the level of abstraction they provide to the user and the extent to which a user is capable of configuring the provided resources. These services can also be seen as a software/cloud stack and depending on this stack these cloud services are often classified as being Software-as-a-Service (SaaS), Platform-as-a-Service (PaaS) or Infrastructure-as-a-Service (IaaS). Software as a Service products are at the top of the cloud stack and provide fully-fletched, ready for use applications, meant to replace locally installed desktop applications. They operate at a high level of abstraction, leaving little to no configuration to the user. The Platform as a Service model on the other hand, offers no applications but rather a toolkit to its users, that can be used for developing, testing and deploying applications that can leverage cloud features. With PaaS products, the user does not necessarily know how much processing power or memory they have available. Cloud services known as Infrastructure as a Service provide these details, letting the user choose and manage their own virtual machines (VMs), equip them with computing and storage resources (virtualized CPUs, memory, hard drives) and install and run custom software [82, p. 13–16]. A more detailed taxonomy of cloud services is given in [69]. Figure 2.1 visualizes the cloud computing stack graphically.

In addition to the categorization based on the different functionalities cloud providers make available, they can be categorized by the deployment model they offer. A cloud can be classified as being *public*, *private* or something in-between often referred to as *community*, or *hybrid* cloud. Public clouds are large, multi-tenant systems that are used

concurrently by multiple users, possibly residing in different locations around the world. Their hardware is generally hosted on the service provider's premises. Usually, they are open to the general public and offer their utilities on a pay-per-use model. Private clouds are for large cooperations and private businesses as well as government agencies. They are built to support their businesses operations and while they can be accessed from different locations, access to them is only granted to certain people. The hardware of private clouds is often times hosted on the organizations' premises or within one of their data center(s). If privacy issues are a main concern, a company may establish a private cloud instead of opting for a public one. Community clouds have similarities with both public and private clouds. They are shared by a community of consumers that have shared goals or concerns. The infrastructure is available only to those users and is owned, managed and operated by one of the entities from the community and or a third party. Finally, hybrid clouds are a composition of two or more private, community or public clouds, that are connected by a shared application or proprietary technology. The individual clouds are hosted by their respective hosts and remain separate entities [59].

2.1.2 Fog Computing

In Section 2.1.1, we argued that one of the characteristics of Cloud Computing is a shared pool of computing resources. Although not stated explicitly, these resources are thought to reside inside large data centers or anywhere physically further away from where the actual computing and storage capabilities will eventually be used.

With the advent of smart devices and the introduction of the Internet of Things (IoT) in fields such as health care, home and public environments, new challenges are faced. Shi et al. identified the following issues [72]: First, *large quantities* of data are generated from various devices and the transmission link to the cloud is becoming a bottleneck. Second, the *response time* from the cloud, which is related to the fact that large quantities of data have to be sent over the network, may be too high for latency critical applications. And third, privacy protection is an obstacle to Cloud Computing, because IoT devices may record confidential information and users may not be willing to put their data on an outside server.

One way of tackling those issues, is performing storage and computation outside of data centers and in closer proximity to the devices and/or end-users that need it (Fog Computing). This is best illustrated by the hypothetical scenario of a child that has gone missing in a city [72][1]. Today, many public spaces have surveillance cameras, and their recordings are often stored (at least temporarily) on a server. Also, many vehicles and smartphones have cameras installed. Usually, these pictures and videos are not uploaded to a server, due to capacity constraints and privacy concerns. With the fog computing paradigm, however, these devices could work together. When a child goes missing, the authorities could distribute that child's' photo along with a request to all

[1] The researchers in [72] actually use the term *Edge Computing* in their publication, but they refer to the same architecture/vision, that we do. For consistency, we chose to use the term Fog Computing throughout the whole text.

devices in a target area to search in their storage for that child. When one device finds the child in one of its recordings, the result is reported back and an investigation can use this information to find it. In this scenario, network overhead is reduced and privacy concerns of individual users are addressed.

Summarizing, Yi et al. provide the following definition of Fog Computing [87]:

> *Fog computing is a geographically distributed computing architecture with a resource pool [that] consists of one or more ubiquitously connected heterogeneous devices (including edge devices) at the edge of network ... [to] collaboratively provide elastic computation, storage and communication ... to a large scale of clients in proximity.*

2.1.3 Edge Data Centers and Users

In order to make fog computing reality, additional hardware will be needed. If, say, a city wants to leverage fog computing, instead of having one big data center, it must spread and deploy 10s-100s of smaller *edge data centers* (EDCs) over its area. One of the applications that would be enabled this way, would be medical cyber-physical systems, a recent and growing trend in healthcare, that allow the efficient usage of home medical monitoring devices [36].

EDCs consist of one or more heterogeneous machines, that are in close proximity to the user such as in radio base stations or WiFi access points. They operate at different costs and provide higher bandwidth and lower latency than their centralized cloud counterparts. They can perform analytics and aggregation tasks immediately after the data were captured. Furthermore, they are location-aware and allow to flexibly add or remove network functions based on the current demands of nearby users [52].

As users are located in or move through the city, they can connect to one of the EDCs within their vicinity. Some coordination mechanism between EDCs has to be in place in order to assure that users connect to the 'right' EDC and do not overrun one base station, while other basestations have almost no users.

2.1.4 Virtual Machines

VMs have become an important tool in cloud computing as they provide the means of transferring user applications from one physical machine to another. This is done by moving whole VMs, instead of individual tasks or processes. VMs offer a complete execution environment to user applications, thus separating them from the actual underlying physical server (the *host*)[2]. Moving VMs, also called *VM migration*, is facilitated by the fact that the interfaces of many server hardware platforms that are used by VMs are well-defined, thus making it possible to run a single VM on different platforms from

[2]Here, we focus the discussion on *system VMs*, that provide a complete and persistent system environment, capable of running several processes at the same time. *Process VMs*, on the other hand, are created and terminated for the use of a single process only.

different vendors [75]. Migrating a VM together with all its applications means that any applications that are dependent on their environment, do not have to be re-written or otherwise adapted, since the execution environment is the same before and after the migration process. As a result, these applications (together with their VMs) become location independent.

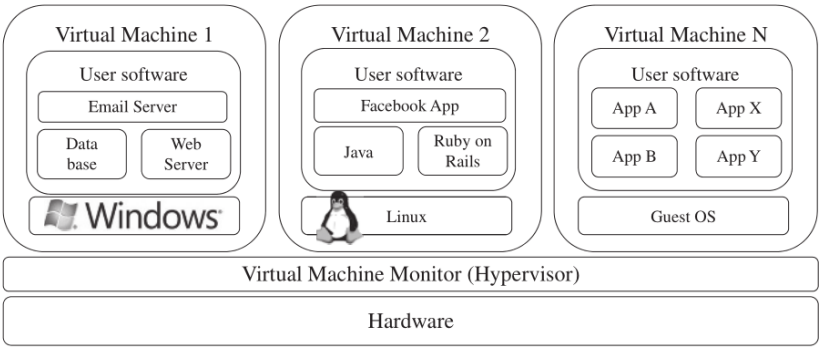


Figure 2.2: A server containing three VMs, each of which run different operating systems and different user applications. A Virtual Machine Monitor or *Hypervisor*, provides a hardware abstraction and runs the VMs [82, p. 10].

Another reason why VMs are popular in cloud computing is that one physical server, is able host multiple VMs at the same time (see Figure 2.2). However, this is only possible, as long as the sum of the resource demands of all the hosted VMs do not exceed the servers' resources. Resource demands of VMs are not always static but change over time, making it difficult to predict how many VMs a server should host. Examples of such resource demands are: processing power, memory usage, network bandwidth (download/upload) and file I/O operations (disk usage). A servers' CPU can only compute a finite number of instructions per second, a servers' network controller is only able to send and receive a certain number of packets per second and hard disks do not spin with infinite speed. If a server is overwhelmed and cannot fulfill the desired resource demands of its VMs, user applications running on these VMs will be experience performance degradation or even come to a complete halt.

2.1.5 Workloads

Workloads can be a result of user requests to a web server, can be due to an application that queries a database server or a periodically running batch job, that does a regularly scheduled backup of some files. We will use the term workload, regardless of whether it affects a servers' CPU, memory or I/O components.

We will focus on transferring one or more VMs from one server to another as the main mechanism for shifting or distributing workloads. One of the goals in cloud computing is to pack as many VMs as possible onto a single server, in order to shutdown the rest of the servers or put them in idle mode. This is known as *VM consolidation*. In idle mode,

servers consume significantly less energy. Therefore, VM consolidation can be seen as an important tool for reducing energy consumption and saving costs.

Workload distribution has to be done carefully. If one underestimates the future workload of a business-critical e-commerce application, for example, and because of that, *underprovisions* resources, a company might have a disadvantage over its competitors and loose customers, if there is a sudden increase in workload. Amazon found, that every 100 ms increase in loading times of their website, costs them 1% in sales, because their customers are more likely loose interest and stop shopping [1].

2.1.6 VM Migration Mechanisms

VM migration can be done in several ways. A naive approach would be, to stop the original VM, copy all of its contents to the desired destination and then continue operations from there. This process is called *pure stop-and-copy* and although it may be simple to implement, there are better approaches in terms of migration length.

The length of moving a VM from one physical host to another is usually measured in *downtime* and *total migration time*. Downtime is the period during which the services offered by the VM are not available, due to there being no currently executing instance of the VM. Clients will notice this as a service interruption. Total migration time is the period from when the migration process was first initiated, until the last piece of memory content was transferred and the original VM is removed. Both downtime and total migration time should be minimal.

The process of memory transfer can be split-up into three phases [27]:

Push phase Pages are pushed across the network from the old to the new destination. During this time, the source VM continues its normal operation. Pages that are modified (made *dirty*), must be sent again.

Stop-and-copy phase The source VM stops its operations, while pages are still being transferred. The VM at the new location is started.

Pull phase The new VM starts its operations. If it tries to access a page that has not been transferred yet, a page fault occurs and the required page is *pulled* across the network from the source VM.

Most practical solutions use only one or two of these three phases. The naive approach described above, uses only the second phase. However, this leads to a downtime and total migration time that is proportional to the memory allocated by the VM.

Figure 2.3 shows a migration timeline that uses the first and second phase called *iterative pre-copy migration*. After initial setup steps (Stage 0+1), all pages of the source VM are copied to the target location and pages that have been modified while the copy process was ongoing, are transferred again (Stage 2). The source VM is then suspended,

remaining pages are copied (Stage 3). At this point, there are two identical (but inactive) VM images on both machines. The source VM is discarded (Stage 4) and finally the network traffic is rerouted to the new VM, which starts its operations (Stage 5).

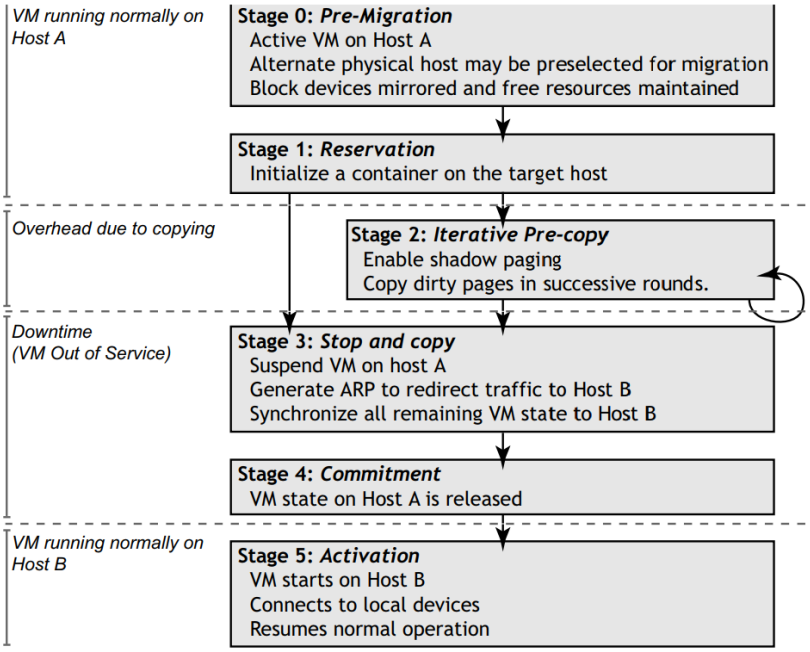


Figure 2.3: Migration process example as described in [27].

This process has a much smaller downtime. However, the total migration time depends on how many iterations have to be made in the pre-copy phase. This in turn, depends on how frequently and how many pages are changing between each round (*page dirtying rate*). Details on how to deal with this issue are discussed in [27].

2.1.7 Remarks

In Section 2.1.4, we discussed that *load* of a server and by extension a servers' *energy consumption* are important perspectives, when reasoning about VM assignment. These perspectives mostly concern the cloud provider, because energy consumption translates to financial costs for them. For cloud users this may be only a minor concern. From their point of view, other issues such as the number of VM migrations, which translates to availability of their services and the financial costs that arise from running their VMs, are more critical. Still, there are many more aspects, that could be taken into consideration, when designing and implementing an ideal cloud service from the standpoint of a user. Kritikos et al. surveyed the literature on cloud computing and identified 43 parameters, that are used to describe the quality of cloud resources/providers [49, Table III]. This

extensive list contains not only performance metrics and availability, but also quality criteria relating to ease of configuration, security and reliability.

In Section 2.1.5, we implied that putting a server that has currently no VMs assigned into idle mode is the only means of saving energy and costs. It should be noted, that modern CPUs are capable of automatically adjusting their frequency. That means, that if a servers' CPU is not fully utilized, it does not consume full energy. If a low processing workload is predicted for a specific server, because its VMs are mostly memory-bound, the CPUs' frequencies will be throttled through *Dynamic Voltage and Frequency Scaling* (DVFS). This results in a significant reduction of energy consumption and reduces the amount of heat that is produced and also saves costs [51]. Conversely, if a high processing workload is impending, the voltage of one or more CPUs can be scaled up.

2.2 Networking

2.2.1 Software Defined Networking

In modern cloud and fog computing environments, not only are the computing resources virtualized, but also the network infrastructure. Instead of dealing with physical links and physical switches, we are dealing with *virtual* links and *virtual* switches. *OpenFlow* [58] is an important communications standard that enables virtualization and serves as the foundation of many other networking approaches, such as *ElasticSwitch* [66]. Virtualized networks offer benefits that conventional networks do not. Most virtualized networks allow its tenants to demand a *minimum bandwidth guarantee*. This is very useful to the tenants as their networked applications often rely on network throughput and this enables them to derive a performance guarantee of their applications. We will describe *OpenFlow*, a technology that enables network providers to configure their infrastructure in a fine grained manner, and *ElasticSwitch*, another technology, that builds upon OpenFlow, to enable bandwidth guarantees in these networks.

2.2.2 OpenFlow

OpenFlow [58] was first introduced in 2008 and its current version 1.5.1 is available under [13] (released March 2015). OpenFlow is an open networking standard that describes a list of criteria that Ethernet switches (or routers) have to meet. It is intended to be vendor independent in order to enable researchers to program switches and run experiments using a unified programming interface. To be OpenFlow complaint, switches must support: (1) A *Flow Table*, with an action associated with each flow entry, that are carried out when a certain flow is encountered; (2) a *Secure Channel* that connects the switch to a controller, that can configure that switch, e.g. adding and removing flow entries, using (3) the *OpenFlow Protocol* which specifies the common language understood by both switch and a controller. The controller is thought to reside in the same network and acts as a means of configuration for researchers. An OpenFlow Switch is visualized

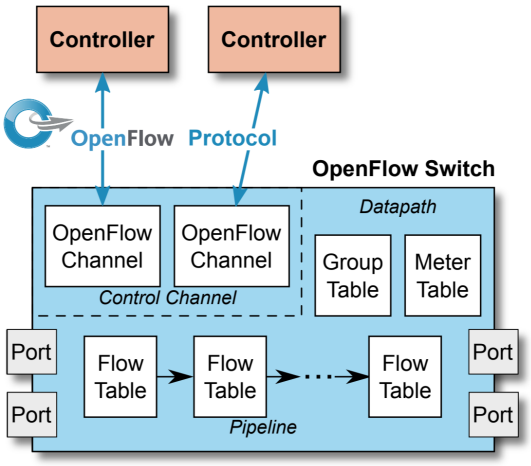


Figure 2.4: OpenFlow visualization: OpenFlow Switches contain a Flow Table that manages forwarding of flows. Flow Tables can be configured using one or more controllers, that establish a secure connection to the switch and use the OpenFlow protocol [13].

in Figure 2.4. We will briefly discuss the structure of the Flow Table and the four actions that it has to offer.

OpenFlow uses the abstraction of a *flow*. A flow can be, for example, all network packets originating from a certain MAC address or IP address, a TCP connection or all packets containing a certain port number. Each Flow Table entry has an action associated with it, which can be one of the following options [58]:

(1) Forward this flow's packets to a given port (or ports).

(2) Forward this flow's packets to a controller.

(3) Drop this flow's packets.

(4) Ignore this flow's packets by forwarding them through the switch's normal processing pipeline.

If a switch offers the capabilities (1)-(3), it is called *Dedicated OpenFlow switch*. If a switch additionally offers capability (4), the standard calls it *OpenFlow-enabled* or *Type 0* switch. The fourth capability is required when one wants to separate experiment traffic from normally occurring network traffic. Given that capability, researchers can run isolated experiments, without interfering with other network users.

This framework, although conceptually easy to grasp, has some limitations in practice. Guo et al. note, that there are practical limits to the maximum size of a flow table. These limits can be reached under normal circumstances quite rapidly, even in a medium sized data center, because of the fact that aggregation switches need to store a lot of forwarding entries, while having only limited memory to do so [38].

Regardless of this limitation, there are several use cases that OpenFlow enables: For example, OpenFlow can be used to implement network management and access control. A controller analyses each new flow using a set of pre-defined rules such as "Guests can communicate using HTTP, but only via a web proxy" or "VoIP phones are not allowed to communicate with laptops" [58]. Another example is *virtual LANs* (VLANs): OpenFlow can provide users (or VMs) with their own isolated network. This can be realized by identifying flows from a specific user (or VM) utilizing the flow table and the users' MAC address, and successively tagging these flows by assigning a VLAN ID to them.

An application that uses the OpenFlow protocol for configuration is *Open vSwitch* [64]. Instead of physical switches, Open vSwitch provides network abstraction to VMs using *virtual switches*. They are implemented entirely in software and run on a servers' hypervisor. In this way, it is possible to implement sophisticated network functions, without relying on switches to offer any non-standard capabilities. Additionally, virtual switches can derive information through integration with virtualization software, that would otherwise be difficult to obtain from inspection of network traffic alone. For example, they can determine all the MAC addresses of all the virtual interfaces belonging to a single VM, whether or not a virtual interface is in promiscuous mode or what IP addresses are currently assigned to a specific virtual interface [64].

2.2.3 ElasticSwitch

As already mentioned above, VM operators would like to have network bandwidth guarantees from the infrastructure service provider, in order to estimate the performance of the applications running on their VMs. One way for service providers to derive these guarantees is through *static reservation* of bandwidths to their customers. However, there is a major shortcoming of this approach: Consider two customers A and B share the same network link and have some pre-assigned fixed capacities. A and B are both transmitting data. When A is not fully utilizing its assigned share, B could use A's residual capacity. Through a static reservation scheme, this is not possible, resulting in longer than necessary transmission times for B. Also, the network infrastructure is not used to its fullest potential which is wasteful from an infrastructure providers point of view. Matters are even worse, when considering that network traffic is bursty in nature and average utilization in networks is low [26, 44].

ElasticSwitch [66] provides bandwidth guarantees to VMs via a more refined reservation scheme by strategically rate-limiting VMs using Open vSwitch, therefore running entirely in hypervisors and requiring no support from switches.

ElasticSwitch relies on the *hose model* of networking first introduced in [30]. Therein, a network link is modeled as an undirected edge with a single value for both ingress and egress bandwidth (therefore *symmetric*) [71] and every physical machine is connected to every other physical machine via a single virtual switch. Once a tenant submits a VM and a bandwidth requirement (using the hose model), ElasticSwitch works in two steps. The first step is called *Guarantee Partitioning* (GP). It transforms the hose model

guarantees of every VM into a set of absolute minimum guarantees for each pair of VMs that need to communicate. More precisely, the guarantee of VM X and VM Y, $B^{X\to Y}$, is set to the minimum of the guarantees assigned by either hypervisor ($B_X^{X\to Y}$ and $B_Y^{X\to Y}$) for the traffic between X and Y

$$B^{X\to Y} = \min(B_X^{X\to Y}, B_Y^{X\to Y}). \tag{2.1}$$

The second step is called *Rate-Allocation* (RA), takes the minimum guarantee of the previous steps as an input. RA tries to use up the additional capacity of any not congested network links: Between every pair of source and destination VM, a rate-limiter on the host of the source VM limits the traffic originating from that source VM to a certain value $R^{X\to Y}$, but not lower than $B^{X\to Y}$:

$$R^{X\to Y} = \max(B^{X\to Y}, R_{W\_TCP}(B^{X\to Y}, F^{X\to Y})) \tag{2.2}$$

where R_{W_TCP} is the rate given by a weighted TCP-like algorithm operating with weight $B^{X\to Y}$ and congestion feedback $F^{X\to Y}$. Intuitively, $R^{X\to Y}$ increases and decreases proportionally to how much congestion is experienced (using a simple weighting scheme that is influenced by $B^{X\to Y}$). Congestion is measured by the amount of packet loss.

GP and RA are executed periodically to adjust bandwidth guarantees according to current demands. It is important to note that ElasticSwitch is orthogonal to the VM placement strategy as long as *the sum of the bandwidths guarantees traversing any link L is smaller then L's capacity* [66].

2.2.4 Topology Generation and High Level Internet Structure

We have discussed how the mechanics of virtualized networks work (on a conceptual level) in the previous Section, but did not discuss the network architecture or topology. In order to mirror the Internet's architecture and the way individual nodes (PMs) are connected with each other, we need a model that describes its key characteristics. Many models for the topology of the Internet have been proposed in the past [40]. The *Positive-Feedback Preference* (PFP) model [93] is one of the more recent ones and was introduced by Zhou et al. in 2005.

Three important empirical observations have been made in the past about the Internet's topology at the Autonomous System level. First, the probability $P(k)$ that a node is connected with k other nodes, decays as a *power law* function

$$P(k) \sim k^{-\gamma}, \tag{2.3}$$

where γ is some constant. This was first realized by Barabasi and Albert [22], when they analyzed the connectivities of various large networks. In their landmark publication, they could show that this relation could be observed in various of fields of applications as well, see Figure 2.5. If a networks' connectivity follows Equation 2.3, the network is called *scale-free*. Zhou et al. note, that this formula does not always hold true, because

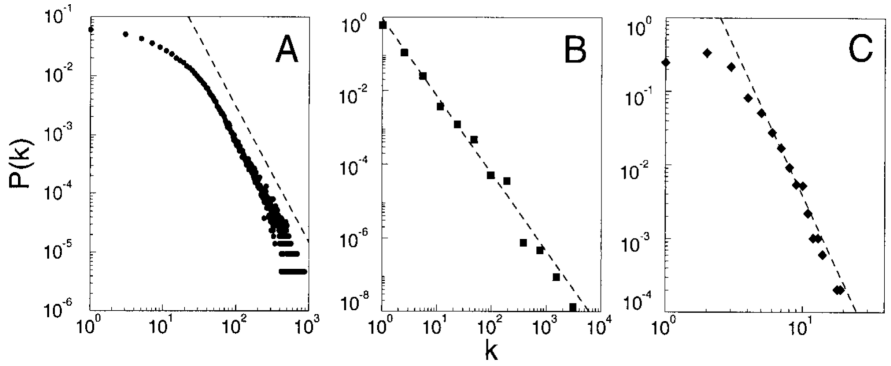


Figure 2.5: Three different network graphs and the degree of their nodes from [22]. A) Actor collaboration graph (212,250 nodes), B) Part of the WWW (325,729 nodes) C) Power grid data (4,941 nodes).

there are some important graphs which consist of nodes with degree 1 than nodes with degree 2, $P(k = 1) > P(k = 2)$ [93].

Second, the Internet expresses *disassortative mixing behavior*. In a general, network theory context, this means that nodes which are *not* similar, connect with each other. Regarding the Internet's topology, it indicates that high-degree nodes have a large number of low-degree neighbors.

Finally, the third observation is called the *rich-club phenomenon*: Nodes with a high degree (the "rich" nodes) are more likely to be well connected with other high-degree nodes, forming a tightly connected clique [92].

2.2.5 The PFP Model

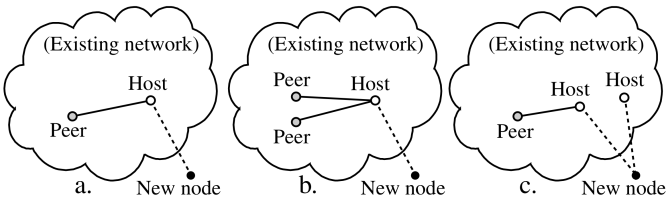


Figure 2.6: Explanation of graph generation using the Positive-Feedback Preference model [93]

The PFP model provides a simple algorithm for generating networks that comply with the three observations mentioned in Section 2.2.4. The algorithm starts with a small random network and in each iteration the network grows by executing one of the three scenarios depicted in Figure 2.6.

1. A new node is attached to one host node. One new internal link appears between the host node and another existing node.

2. A new node is attached to one host node. Two new internal links appear between the host node and two other existing nodes.

3. A new node is attached to two host nodes. One new internal link appears between one of the host nodes and another existing node.

The probability for each of the scenarios is $p \in [0, 1]$, $q \in [0, 1-p]$ and $1-p-q$, respectively. The researchers suggest to set $p = 0.3$ and $q = 0.1$, as this produces the same ratio of nodes to links as a well-known Autonomous System graph. One of the strengths of this algorithm is that it implements all the three observations mentioned above really well, especially the rich-club phenomenon. In fact, as the graph grows, the rich nodes not only become richer, they become disproportionately richer. Also, the model accurately reproduces the correct ratio between nodes with degree 1 and nodes with degree 2. An example of a graph generated by the PFP model is shown in Figure 2.7.

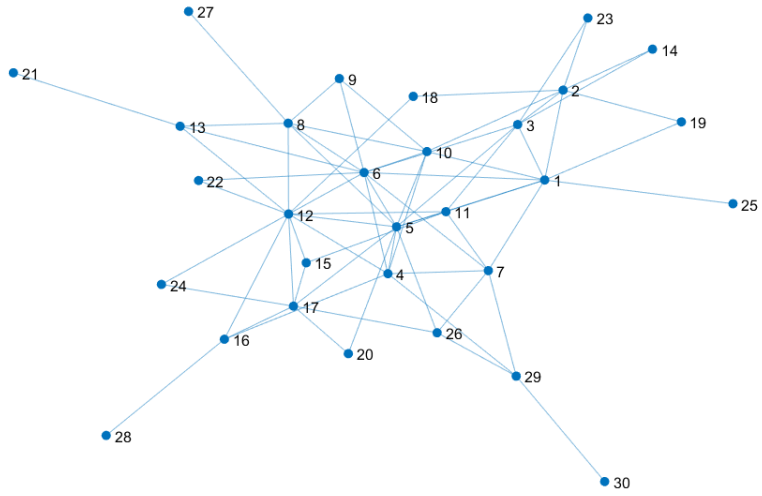


Figure 2.7: Example of a network topology graph with 30 nodes that was generated based on the Positive-Feedback Preference model [93].

2.2.6 Important Graph Measures

In order to compare two graphs, for example, a randomly generated graph and one that was produced by the PFP model, we briefly introduce some important graph metrics, that quantitatively capture some key topological characteristics, that are related to important network dynamics. We will later use these metrics in Section 5.4.5 for evaluation purposes.

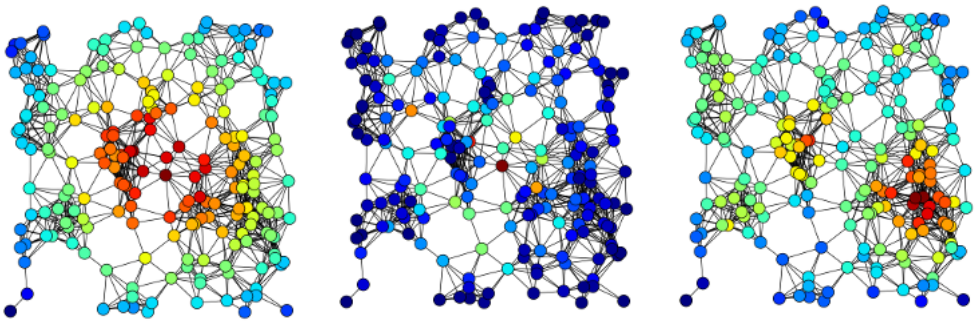
Let $G = (V, E)$ be a graph, where V denotes the set of vertices $v_1, ..., v_n$ and E denotes the set of edges $e_1, ..., e_m$ in the graph. Let *eccentricity* of a vertex v be the length of the longest shortest path from v to any other vertex v' in V with $v \neq v'$. Intuitively, reaching

all other vertices from a vertex with a high eccentricity takes longer, than from a vertex with a low eccentricity. In a disconnected graph, all vertices have infinite eccentricity [83].

Mean Shortest Path Length is the average length of all shortest paths in the graph. Low values indicate that on average the nodes in the graph are close to each other.

Graph Radius is the minimum eccentricity among all vertices in the graph.

Graph Diameter is the maximum eccentricity among all vertices in the graph.



(a) Closeness Centrality **(b)** Betweenness centrality **(c)** Degree centrality

Figure 2.8: Different centrality measures applied to same graph [18].

Let *centrality* capture the importance of a vertex u [11]. Three different centrality measures will be used: *Closeness* centrality of a vertex u is the inverse sum of the shortest path lengths $d(.,.)$ from u to all other vertices in the graph.

$$c_{close}(u) = \frac{1}{\sum_v d(v,u)} \tag{2.4}$$

Betweenness centrality measures how often each vertex appears on a shortest path between two vertices in the graph. Let $n_{st}(u)$ be the number of shortest paths from vertex s to t that pass through u and N_{st} be the total number of shortest paths from s to t. Then, betweenness centrality of vertex u is defined as

$$c_{between}(u) = \sum_{s,t \neq u} \frac{n_{st}(u)}{N_{st}} \tag{2.5}$$

Degree centrality of a vertex u is the number of edges connecting to u. Self-loops count as two edges connecting to the vertex. For simple graphs, this measure is the number of adjacent nodes to a node u.

$$c_{degree}(u) = deg(u) \tag{2.6}$$

We will use **Mean Closeness Centrality**, **Mean Betweenness Centrality** and **Mean Degree Centrality** of all vertices in G as additional graph measures.

2.3 Biogeography-Based Optimization

The Biogeography-based Optimization (BBO) algorithm was first published by Simon in 2008 [73]. It is a type of evolutionary algorithm that uses generations of candidate solutions to find the global optimum (either minimum or maximum) of a real-valued objective function. The idea for this algorithm comes from the observation how species in nature migrate between their natural habitats and how new species arise and how existing species become extinct. In the following section we detail the algorithm, and describe later how it was adapted for our specific problem.

2.3.1 Algorithm Terminology and Procedure

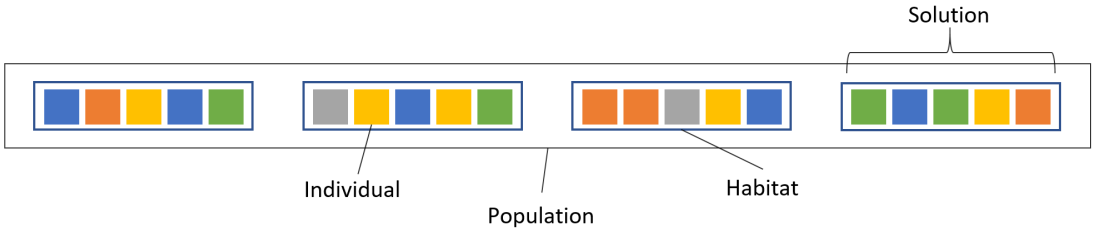


Figure 2.9: Biogeography-Based Optimization Terminology

Given a multidimensional, real-valued function $f(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}$, we want to find $\mathbf{x}$ such that, $f(\mathbf{x})$ is at its global optimum (either minimum or maximum). $f(\mathbf{x})$ is often called objective or *fitness function* of $\mathbf{x}$ and is seen as a measure of quality of $\mathbf{x}$. $\mathbf{x}$ is called the (candidate) *solution*.

The algorithm uses an iterative approach to search through the space of all possible candidate solutions. It consists of the following major steps:

- Generate an initial population of solutions for the given problem via some procedure.
- While a termination condition is not met, do the following:
  - Rank the population based on their fitness function.
  - Save the best solutions temporarily.
  - Alter the population through *immigration* and *mutation*.
  - Again, rank the population based on their fitness function.
  - Replace the worst solutions, by the saved, best solutions from before.

We briefly discuss the terminology used. In each iteration, it creates and updates a set of candidate solutions, which is called a *generation*. The number of solutions per generation

is chosen beforehand. The larger the number, the more thoroughly the search space is explored, but also the more computationally expensive the procedure becomes.

Let S_{max} be the number of solutions per generation. The objective function is evaluated S_{max} times, once for each solution, and based on the result the solutions are ranked from best to worst. The top k solutions are temporarily saved. These solutions are called *elites*.

A solution $\mathbf{x} = [x_1, x_2, ..., x_n]$ is also called a habitat or *island*. The best solution is the habitat that is the most *suitable*, the worst solution is the least suitable. The constituents of a solution $x_1, x_2, ..., x_n$ are also called independent variables or *individuals*.

The migration step is inspired by nature. In the migration step, individuals are allowed to roam freely, meaning that they can migrate from their original habitat to another one. Individuals currently living in very suitable habitats, prosper and proliferate and therefore tend to leave it. When an individual decides to migrate, it replaces an individual in the other solution, which has to move out (*emigrate*). This other individual is then extinct.

After the immigration/emigration procedure has been completed, random *mutation* takes place. Again, this is inspired by nature and how animals procreate. That is, some of the individuals are replaced by randomly generated individuals from the allowed range of values. This step creates new solutions and allows the algorithm to explore more of the search space. Next, the objective function is evaluated again for each solution and the worst k solutions are replaced by the previous elites. Finally, this new set of solutions constitutes the next generation of the algorithm, which then starts again. The algorithm terminates after a fixed number of generations has been produced or the objective function has been evaluated a predetermined number of times.

The details of initialization are specific to the problem at hand, and will be discussed in Section 4.3.1; ranking, migration and mutation will be the focus of the Sections 2.3.2, 2.3.3 and 2.3.4, respectively.

2.3.2 Ranking of Solutions

The original version of BBO uses a scalar-valued objective function. Ranking based on a scalar-valued function is straightforward. For our specific problem, however, we are using vector-valued objective function $\mathbf{f}(\mathbf{x}) : \mathbb{R}^n \mapsto \mathbb{R}^q$ (explained in Section 4.2), which forces us to adapt the original ranking process. The new ranking process is as follows.

Let $\mathbf{f}(\mathbf{x}) = f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_q(\mathbf{x})$ be a vector-valued function and assume, that we want to find the minimum of all functions. Let $S = [x_1, ..., x_{S_{max}}]$ be the generation, that needs to be ranked, let $R = []$ be a list that represents the final ranking, and let $i = 1$ be a counter variable.

1. Get the set of solutions σ among S for which f_i is minimal among all solutions.
   a) If $|\sigma| = 1$, go to Step 2.
   b) If $|\sigma| > 1$, then iteratively cull the set by selecting those solutions of σ for which f_j is minimal among all solutions, where $j = [2..q]$ if $i = 1$ and $j = [i+1.., q, 1..i-1]$ if $i > 1$. If after this procedure σ still contains more than one solution, choose a solution arbitrarily from σ and remove all others.
2. Add the single solution s of σ to R, remove s from S and set $i = ((i+1) \bmod q) + 1$.
3. If there are still solutions left in S, go to Step 1. Otherwise, terminate.

The result of this process is, that the first q solutions in R are the best ones for each of the objective functions of $\mathbf{f}(\mathbf{x})$, the next q are the second best solutions for each of the objective functions, and so on. If two solutions x_i and x_j are tied with respect to a certain objective function, $f_p(x_i) = f_p(x_j)$, then the next function is used as a tie breaker (wrapping around to the first function, if f_p was the last function). So, good performing solutions are at the head of list R, worse solution are at its tail. Table 2.1 depicts the ranked solutions. Based on this ranking immigration and emigration rates of the solutions' individuals are determined (see Section 2.3.3).

Table 2.1: Fitness-based immigration and emigration rates. Solutions are assigned values λ_k and μ_k based on their relative fitness to other solutions in their generation.

Rank	**Solution**	λ	μ
1	x_{r_1}	λ_1	μ_1
2	x_{r_2}	λ_2	μ_2
3	x_{r_3}	λ_3	μ_3
...	...	...	...
k	x_{r_k}	λ_k	μ_k
...	...	...	...
S_{max}	$x_{r_{S_{max}}}$	$\lambda_{S_{max}}$	$\mu_{S_{max}}$

2.3.3 Migration

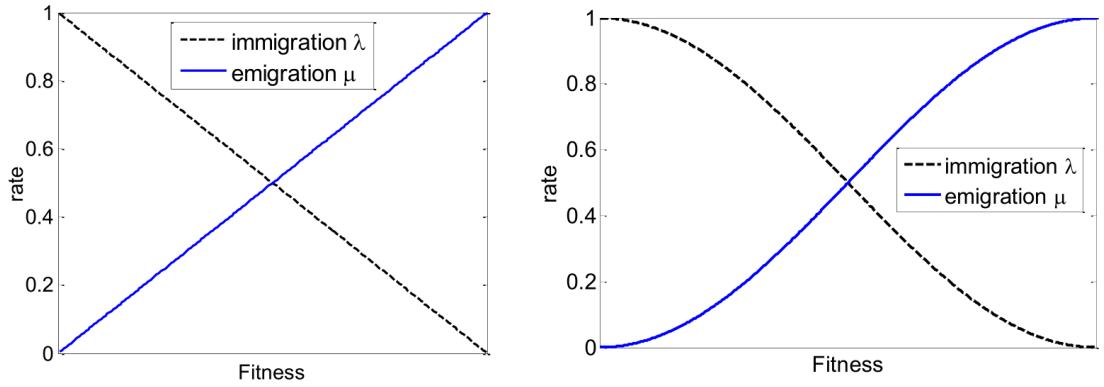


Figure 2.10: Linear (left) and sinusoidal (right) immigration and emigration functions [46].

In Section 2.3.1, we discussed migration of individuals between islands. Migration occurs randomly according to the immigration rate λ and the emigration rate μ, that are specified before the algorithm is started. Figure 2.10, shows two examples of these rates, linear (left) and sinusoidal (right), where $\lambda_k = 1 - \mu_k$. These immigration and emigration curves can be more complicated than these examples (see [57] for a detailed performance analysis of six different migration models). Nevertheless, they provide a sufficiently good approximation for our purposes. As we mentioned in the previous section, notice, that the rates are a function of the fitness of the solutions.

The better a solution, the *higher* the emigration rate of its individuals. This might sound counter-intuitive at first. However, we want the parts of the good solutions to spread widely among the generation, in order to produce even better solutions in the next generation. The immigration rate for a good solution is small, because we do not want to destroy already good solutions.

The worse a solution gets, the *lower* is its emigration rate. A similar argument can be made here: We want the parts of the bad solutions extinguished and replaced by other parts that are known to yield better solutions. The emigration rate for a bad solution is small, because we do not want to preserve bad individuals for future generations.

We now have an intuition for how migration takes place. Let us now discuss the immigration/emigration step more formally. Let x_k be the k-th solution of a generation and λ_k and μ_k its corresponding immigration and emigration rates, respectively. For each independent variable v in x_k do the following: Create a random number $\rho \in [0..1]$. If $\rho < \lambda_k$, then v has been chosen to be replaced. Choose the replacing variable v_e, the emigrant, among all other solutions of this generation according to

$$P(v_e = v_m) = \frac{\mu_m}{\sum_{i=1}^{S_{max}} \mu_i} \text{ for } m \in [1..S_{max}]. \tag{2.7}$$

This means, that individuals for replacement are chosen according to the relative proportions of their solutions' emigration rate. Equation 2.7 is also known as *roulette-wheel selection*.

The equations for μ_k in the graphs depicted in Figure 2.10 are

$$\mu_k = \frac{S_{max} + 1 - r_k}{S_{max} + 1} \tag{2.8}$$

for the linear model and

$$\mu_k = \frac{1}{2}(1 + \cos(\pi \cdot r_k/(S_{max} + 1)) \tag{2.9}$$

for the sinusoidal model where r_k is the rank of a solution, where $r_k = 1$ is the rank of the best solution and $r_k = S_{max}$ is the rank of the worst solution [46]. The formulas were slightly adapted, to account for the fact that we have multiple objective functions, that we do not want to favour over one another: An alternative rank $\overline{r}_k$ is used instead of r_k and calculated as

$$\overline{r}_{k+1} = \begin{cases} \overline{r}_k & \text{if } k \text{ is not divisable by } q \\ r_k & \text{otherwise} \end{cases} \tag{2.10}$$

, where q is the number of objective functions and $\overline{r}_1 = r_1$. For example, if $q = 4$, the first few $\overline{r}_k$ are $1, 1, 1, 1, 5, 5, 5, 5, 9, 9, 9, 9, \ldots$. Calculating $\overline{r}_k$ therefore ensures that immigration/emigration rates are the same for equally good solutions that are ranked below each other by the procedure described in Section 2.3.2.

2.3.4 Mutation

Mutation is performed to create new solutions, based on existing ones to avoid getting stuck in local optima of the objective function, and to explore the search space of all solutions more thoroughly. In the original version of the algorithm, the mutation step is performed after the migration step, regardless of the quality of the solution (both good and bad solutions have a chance of mutating) and regardless of the number of generations that have been produced by the algorithm (the probability for mutation in one iteration is the same as in the next iteration). We left the original mutation step unchanged, but introduced an additional mutation step, after the first one. This new step is dependent on both the quality of the solution and the number of iterations that have passed.

The original mutation procedure works by (1) selecting variables and (2) replacing that variables: (1) Let x_k be the k-th solution of a generation. For each independent variable v in x_k do the following: Create a random number $\rho \in [0..1]$. If $\rho < p_{\text{mutation}}$, then v has been chosen to be replaced. p_{mutation} is a parameter that is chosen at the start of the algorithm. (2) How the actual replacing variable is formed, is problem specific. In the original, *continuous* version of the algorithm, the investigator has to know a-priori the approximate interval the global optimum is located. Replacing a variable was

implemented by choosing a variable at random from that interval. In Section 4.3.2, we discuss a replacing strategy for our specific, discrete problem.

Our newly introduced mutation step relies on two parameters: $q \in (0, 0.5)$ and $\gamma \in [1, 20]$. The first one, determines how many members of the current generation should be chosen for mutation. For example, if $q = 0.25$, then the worst 25% of solutions are selected. After the solutions have been chosen, T is calculated that determines how many individuals should be mutated in every solution. Equation 2.11 shows the formula for deriving T.

$$T(i) = \begin{cases} e^{-i\gamma} & \text{if } e^{-i\gamma} \leq 0.5 \\ 0.5 & \text{otherwise,} \end{cases} \tag{2.11}$$

where i is the count of how many iterations the algorithm has undergone. Then, $\max(1, \lfloor nT \rfloor)$ individuals are mutated, where n is the length of a solution. In other words, roughly T% of the variables are mutated.

Figure 2.11, shows how the exponential function behaves in the interval $[0..1]$. The higher the parameter γ, the more negative the exponent becomes and the more rapidly the function approaches zero. This entails, that in the first algorithm runs, a lot of mutations will take place and because the worst solutions are changed drastically. As more and more iterations have passed, only a few variables will be mutated. This is desirable, because at first the algorithm has not converged and much optimization is possible. Later, the algorithm has (hopefully) converged towards an optimum and we want to avoid changing too many variables of the already near-optimal solutions. The idea for this iteration-dependent parameter, stems from *Simulated Annealing* [47].

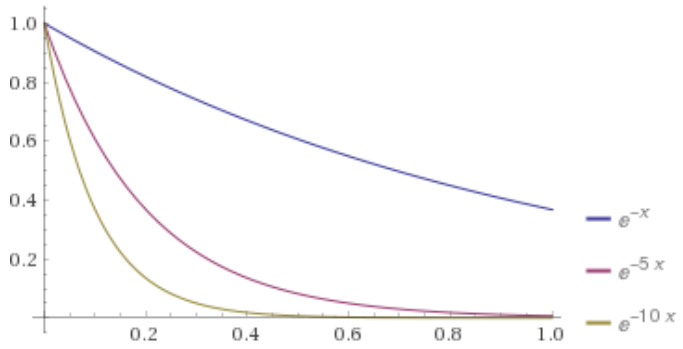


Figure 2.11: Different exponential functions $f(x) = e^{-\gamma}$, with $\gamma = 1, 5, 10$.

2.3.5 Further Algorithm Details

Duplicate reduction Since generations are created based on a random process, two identical solutions can be created by chance. In order to reduce the number of duplicates, two additional algorithm steps are employed, that are executed after every mutation run: First, the entire population is scanned for duplicates. Then, if two identical solutions are

found, a random individual of one of them is chosen and mutated again. This process does not guarantee that all duplicates are eliminated. It may happen, that the newly created mutation is identical to some other solution. However, this process drastically reduces the number of duplicates and requires only a single pass over the list of solutions of a generation.

Repair of infeasible solutions Creating solutions using migration and mutation may also yield infeasible solutions. In fact, these solutions may also have a better objective function and are therefore more likely to survive and be carried forward into future generations or their objective function may be ill-defined and therefore an ordering would not be possible. To avoid infeasible solutions, a modification to the algorithm is made. Before the objective function of a solution is evaluated, the solution is checked for feasibility and if it is not feasible, it is repaired. Repairing is problem-specific and discussed later in Section 4.3.3.

2.3.6 Discussion

Biogeography-based optimization is a metaheuristic that has some advantages over other optimization algorithms, that make it a good choice for solving an assignment problem. BBO is a kind of *genetic algorithm* (GA) and as such inherits some properties of it. Any genetic algorithm, does require its objective function $f(x)$ to have any particular shape or to be continuous. In fact, it does not even require $f(x)$ to be differentiable. Other optimization algorithms (such as Gradient Descent [63]) use a gradient to find local optima and are therefore restricted to differentiable functions only [63]. Next, GAs use initial solutions and improve them over many iterations. This has two implications: First, they can be started with a solution, that is already known to be a good solution, to further improve it. This property is also called *forward-compatibility*. And second, at any point in time, these algorithms can be stopped, and the currently best solution can be extracted. This is useful, if, for example, an application needs results earlier than expected. Further, GAs can be parallelized and/or executed in a distributed fashion. For BBO in particular, the migration and mutation phases can be computed simultaneously to gain a performance boost. Some optimization algorithms use and improve a global solution (such as Simulated Annealing [47]) and are not easily parallelized.

BBO has been used in many real-world applications. In its original publication, it was applied to the problem of sensor selection for aircraft engine health estimation [73]. Since then, it has been used in machine learning for classification purposes [45][89], to solve optimization problems such as the traveling salesman problem [76] and to find the optimal shape design of an electrostatic micromotor [29]. Zheng et. al used BBO to find an optimal placement of VMs within datacenters [91]. We will use it to simultaneously find an assignment for both users and VMs in a fog computing environment.

CHAPTER 3

Related Work

In this chapter, we discuss existing solutions to the Virtual Machine Placement (VMP) problem. In the last years, a huge number of solutions were proposed, each of which has its own set of assumptions and subtleties. Looking at the cloud computing realm, in a recent publication, Pires et al. surveyed the existing literature [55] and classified the most recent papers that were concerned with VM placement within a single data center or among two or more data centers. They reference another publication by Beloglazov et al. [24], which identified the following open challenges:

> *(1) development of fast energy-efficient algorithms for the VMP, considering multiple resources for large-scale systems with the ability to predict workload peaks to prevent performance degradation,*
>
> *(2) energy-aware optimization of virtual network topologies between VMs for optimal placement in order to reduce network traffic and thus energy consumed by the network infrastructure,*
>
> *(3) development of new thermal management algorithms to appropriately control temperature and energy consumption,*
>
> *(4) development of workload-aware resource allocation algorithms, considering that current approaches assume a uniform workload, and*
>
> *(5) decentralization and distributed approaches to provide scalability and fault-tolerance to the VMP problem resolution.*

We argue that many of the issues mentioned in the cloud computing literature are also faced in a fog computing environment and therefore these two problems are closely related. Therefore, we review both fog computing *and* cloud computing works. This chapter is structured as follows. First, we introduce some terminology that will allow

us to talk about and classify the different VM placement strategies and give a general overview over the literature. Then, we shortly present two types of commonly assumed architectures for fog computing. Eventually, we discuss some important state-of-the art publications in the field of cloud and fog computing, in particular publications using metaheuristic. In the entire discussion, we will specifically focus on the objective function(s) the researchers chose in their work as well as the solution techniques that were applied.

3.1 Terminology

Before moving on to recent load balancing approaches, we would like to introduce the terminology brought forth by Pires et al. [65]. This will give us the vocabulary necessary to discuss the load balancing approaches more appropriately and concisely.

Algorithms for the VMP problem may be classified by their use of (A) optimization approach, (B) objective functions and (C) solution techniques. One of the following optimization approaches is possible. (1) A *mono-objective* approach (MOP) that considers only one objective function or considers the individual optimization several functions, but one at the time; (2) A *multi-objective solved as mono-objective* approach (MAM) that uses multiple objective functions, combines them into a single function and ultimately solves the problem using only with single function. A popular method for combining objective functions is using a linear combination of them (Weighted Sum Method). Finally, (3) a *pure multi-objective* approach (PMO), that uses multiple objective functions and treats all of them separately. In Section 4.2.1, we explain the concept of Pareto dominance and how this can be achieved. According to [65] a majority of research articles used a mono-objective (61.9%) or multi-objective solved as mono-objective (34.5%) approach to solve the VMP problem and only small percentage actually used a pure multi-objective one (3.6%).

Depending on the optimization approach, the researches can then decide which objective function(s) to use and whether they want to minimize or maximize each of them. Objective functions include functions for energy consumption minimization, network traffic minimization, economical cost optimization (such as economical revenue maximization and SLA violations minimization), performance maximization (such as availability maximization and total job completion time minimization) and resource utilization maximization (such as maximum average utilization minimization and resource wastage minimization).

Finally, the solution technique for the VMP problem is the third classification criterion. The categories are deterministic algorithms (such as integer linear programming), heuristics (first fit, best fit, greedy algorithms), meta-heuristics (ant colony optimization, genetic algorithm, simulated annealing) and approximation algorithms.

3.2 Modeling Approaches and Architectures

In Chapter 1 we explained the architecture that this work relies upon. Other researchers have used other types of architecture that we will briefly discuss here.

A common modeling approach in cloud computing is to use a centralized, global manager, that constantly monitors all the PMs and keeps track of their utilization. This allows the implemented approaches to work with a global system view. All the PMs are viewed as on coherent, yet heterogeneous, cluster of machines. Beloglazov et al. [25], for example, used this modeling approach. Their system model is shown in Figure 3.1.

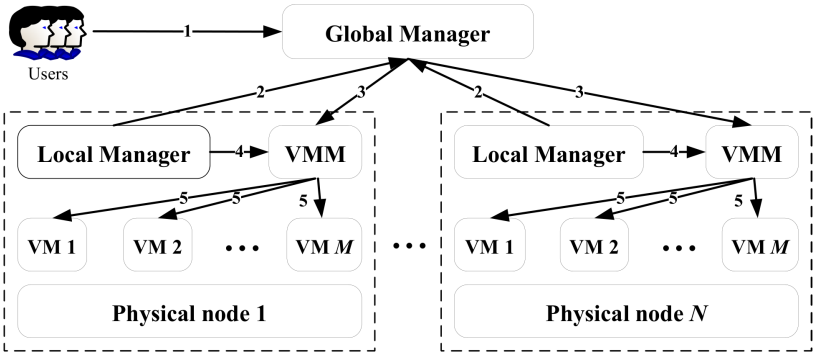


Figure 3.1: System Architecture of Cloud Computing Data Center from [25]

Other researchers take into account that physical machines and data centers may be geographically distributed. This enables to investigate how the availability of the hosted applications is influenced when one or more data centers become unavailable [79]. This can also lead to the development of new controllers that exploit different electricity prices of different locations in order to save costs for data center operators [56].

Gupta et al. [39] use an architecture for fog computing where they explicitly model network link capacities and delays. Their architecture has a tree structure where a cloud data center constitutes the root of the tree, switches form intermediate nodes and sensors and actuators form the leaf nodes. Their application scheduling algorithm can take network traffic into account and avoid congestions, because of that detailed network modeling.

3.3 VM Assignment using Heuristics

Much of the research relies upon the fact that VM assignment can be modeled as *Vector Packing Problem* (VPP) [77], which is a generalization of the classical *bin packing problem* and NP-hard. With classical bin packing, there are a number of items that have a certain size $\in (0, 1)$, and bins that have a capacity of 1. The goal is to find an assignment of items to bins, such that every item is assigned to exactly one bin and as few bins as possible are

used. In the VMP, each item has d sizes, instead of just one, and each bin has d different capacities. This problem was first formalized in [34].

A common heuristic to find a good approximate solution for bin packing is called *Best Fit Decreasing* (BFD). Let OPT be the number of bins that the optimal solution of a bin packing instance needs. Then BFD will use at most $\frac{11}{9} \cdot OPT + 1$ bins [88]. BFD works by first sorting all items that need to be packed in non-increasing order. Each item is then assigned to the bin where it fits the best, that is, the bin for which after the assignment the least amount of space is unused.

Beloglazov et al.[24] analyzed the VM assignment problem in terms of energy usage, which they try to minimize (MOP). They view energy usage as being only dependent on the CPU utilization u of their PMs. The researchers divided the problem into two subproblems: First, finding an assignment of *new* VMs and second, optimizing the assignment of *existing* VMs. The first subproblem, new assignment, can be viewed as bin packing problem with variable bin sizes and prices. To solve it, they devised an algorithm that is a modified version of BFD (MBFD). The second subproblem, optimizing the existing assignment, is tackled in two steps. Selecting those VMs, that need to be migrated and then assigning them to new hosts using MBFD. VM selection is triggered based on two predefined thresholds for u for each PM. An upper threshold which should not be over exceeded, otherwise *one or more* VMs will be migrated away. And a lower threshold, that, when deceeded, causes *all* VMs to be migrated away from the PM. When VM selection is triggered, the researchers defined three selection policies that rely upon (1) minimizing the number of VM migrations needed, (2) selecting the VMs with the highest growth potential and (3) a random choice policy. Beloglazov et al. simulated VM assignment strategy using the *CloudSim* framework, and showed the potential for data center energy savings.

In a publication that same year, Beloglazov et al. [25] again are trying to minimize total power consumption of all PMs (MOP). This time, they use adaptive thresholds, instead of static ones, because, they argue, static thresholds are unsuitable for an environment with dynamic and unpredictable workloads. The adaptive thresholds are calculated based on statictical analysis of historical VM usage data and use *Median Absolute Deviation*, *Interquartile Range*, *Local Regression* and *Robust Local Regression* applied on past CPU utilization values. Again, the CloudSim framework is used for evaluation purposes.

3.4 VM Assignment using Metaheuristics

Now, we would like to narrow our focus to VMP techniques that use metaheuristics. Note that metaheuristics are applicable to a wide array of problems and some of them do not yield a solution right away, but provide more of a *framework* for deriving a solution. This framework may sometimes admit infeasible solutions and we will carefully review, how the investigators deal with this issue of infeasible solutions.

Wu et al. [84] implemented a Genetic Algorithm to assign VMs to PMs. They defined an

encoding scheme for the problem which is identical to ours if we removed the user part from it (see Section 4.1.3). Further, they defined a uniform crossover procedure and a mutation function that, if triggered, changes the assignment of a randomly chosen VM to a random PM. As objective function, they used the sum of energy consumption of both their set of physical machines and network infrastructure (MAM). GA can also produce infeasible solutions. If an infeasible solution is derived by Wu et al., the researchers choose not to disregard it. Instead, they assign the solution a very bad objective function such that it is heavily penalized and made sure that the values of the objective functions of any infeasible solution is less than the objective functions of any feasible solution.

Wu et al. [85] used simulated annealing (SA) and Ali et al. [21] used Biogeography-based optimization to assign VMs to PMs. They both consider only energy consumption of the PMs as their single objective function (MOP) and, again, operate inside a cloud data center and do not consider user assignment. In their evaluations, both papers generate VM sizes (CPU and memory requirements) by choosing a value uniformly at random from a pre-defined interval such that PMs are approximately capable of hosting between 1 and 10 VMs at a time. They consider network bandwidth also as one of the VM sizes, but otherwise ignore network related issues such as migration time of VMs, associated network overhead of migration or routing issues. BBO can create infeasible solution during its operations, but [21] does not mention how they are being treated. [85], on the other hand, mentions that they simply disregard infeasible solutions.

Zheng et al. [91] used BBO for the VMP problem and used multiple objective functions that they treated independently of each other (PMO). They tried to simultaneously minimize: PM resource waste (or, in other words, maximize resource utilization), PM power consumption, PM load unevenness (every PM should approximately have the same utilization), storage traffic and migration cost. VMs are assumed to be communicating with each other and therefore they also minimize inter-VM network traffic.

To minimize these functions, they introduce the concept of *subsystems* to BBO. Within each subsystem, a single objective function is optimized with respect to its own objective and constraints. This means that the solutions of a subsystem are ranked internally and immigration/emigration of individuals only occurs within the subsystem. After that, in a second step, information between those subsystems is shared with each other so that the entire system is optimized. In other words, cross-subsystem migration takes place, where solutions have a certain probability to move from one subsystem to another, based on a similarity metric.

Infeasible solutions are kept in this algorithm: During the within-subsystem migration phase, for each solution, the number of constraint violations is recorded and influence the immigration probability of this solution. Individuals of solutions with fewer or no constraint violations having a higher probability of immigration than individuals of solutions with a high number of violations. If two solutions have the same number of constraint violations, they are further sorted by the quality of their objective functions.

Mennes et al. [60] devised a GA for application placement in hybrid clouds. Hybrid

clouds, according to their definition, are smaller clouds with heterogeneous capabilities that contain unreliable PMs and links that are failure prone. Applications are made up of one or more services and require certain levels of availability guarantees. These services should be deployed in these hybrid clouds according to PM and link failure probabilities. A service may be also be deployed multiple times.

The objective function they maximize is the total number of placed applications (MOP). Their solution encoding or chromosome is vastly different from the solution coding we are using and based on *Biased Random-Key* [23]. Biased Random-Keys have found wide spread usages in metaheuristics [35]. Instead of encoding the solution using integers or a binary encoding, they use a vector of variables from the interval $[0, 1]$. This vector, however, does not readily represent the assignment of applications and services to PM. For that purpose, a decoding function is employed to transform this vector back into the solution space using the vector components as *sort keys* for the decoding function. This effectively diminishes the problem of infeasible solutions, because the decoding function only produces feasible ones.

Gao et. al [33] use *Ant-Colony Optimization* (ACO), another biology-inspired optimization algorithm, for the VM placement problem. They simultaneously minimize total resource wastage and power consumption in a pure-multiobjective approach (PMO) and their procedure yields a non-dominated set of solutions. Ferdaus [32] et al. also use ACO, and minimize the number of PMs that have at least one VM assigned as their single objective function (MOP). Both publications model the problem as multidimensional vector packing problem, using CPU and memory utilization [33], and CPU, memory and network utilization [32] as VM dimensions.

Xu et. al [86] propose a two step approach for VMP. The first is a mapping for workloads to VMs, the second a mapping of VMs to PMs. Here, we will focus on the second step. They minimize resource wastage, power consumption and thermal dissipation using a genetic algorithm. To combine these possibly conflicting goals into one objective function, the researchers use fuzzy logic (MAM): Three fuzzy sets called *small resource wastage* w, *low power consumption* p and *low temperature* t are defined. Then, membership functions of these fuzzy sets are specified as decreasing functions of variable values, since the smaller the variables the better the solution. The result of the weighted-averaging fuzzy operator

$$\mu(x) = \beta \cdot min(\mu_w(x), \mu_p(x), \mu_t(x)) + (1 - \beta) \cdot avg(\mu_w(x), \mu_p(x), \mu_t(x))$$

is the final objective function, with better solutions having a higher $\mu(x)$. β was set to 0.5 and $\mu_w(x), \mu_p(x), \mu_t(x)$ represent the membership functions of solution x of the three fuzzy sets.

Abbasi-Tadi et al. [19] used BBO, however in a different context. The researchers studied task assignment to already assigned VMs, with task makespan as their objective function (MOP). VMs in their scenario are viewed as having power capacities and workload processing capacities, each of which are stored in a global virtual machine monitor and periodically updated via messages. When one or more tasks are assigned, BBO needs to

sort the VMs based on their fitness. Fitness is based on linear combination of required bandwidth, required CPU and required memory of the task(s) scaled by the available bandwidth, available CPU and available memory of each VM, respectively.

3.5 Further VM Assignment Strategies

Ren et al. [68] tackle the VMP problem from an evolutionary game theoretic perspective. According to their problem definition, an application is made up of three VMs (a database server, an application server and a web server). A *strategy* states the locations of and resource allocations for these three VMs. Finding a good strategy is done as follows: The algorithm keeps a population of different strategies for each application. In each population, two strategies are randomly selected and then compete against each other. Competition is based on both feasibility of the strategies and values of their objective functions. The looser is removed from the population and the winner is replicated and mutated with a certain mutation rate. This process is repeated until the population reaches a steady state. Then, the best strategy from each population is chosen, and VMs are allocated based on this strategy. They propose five objective functions (CPU allocation, Bandwidth allocation, Response Time, Power consumption and Workload Distribution) which are considered separately.

Spinnewyn et al. [78] study fault-tolerant application placement. Their applications are made up of services to be placed on a network of error-prone nodes and links. Services can be placed multiple times to increase the availability of an application. Their placement is done in multiple steps, where each uses one of the following objective functions: They try to maximize acceptance, minimize bandwidth usage, minimize CPU resources usage and minimize the number of duplicates used. The authors formulate the problem as a binary Integer Linear Program (ILP) and use the Gurobi Optimizer to solve it [6].

Speitkamp et al. [77] study service assignment in virtualized data centers such that the overall server costs are minimized. They use CPU capacity as their only capacity dimension but consider various other possible constraints to the problem, for example, that a server may only host services up to a maximum number, some service may be hosted only on one specific server and some subsets of services have to be hosted on the same/different servers. The authors use a number of different methods (branch and bound, first fit, first fit decreasing and a linear programming-relaxation-based heuristic) to find approximate solutions for various relaxations of the assignment problem and also study the resulting solution qualities with respect to practical considerations (such as instance size and incorporation of additional side constraints in relation to solution computation time).

3.6 Edge Data Center Load Prediction and Balance

Le Tan et al. [52] used the same architecture in their publication. A number of edge data centers dispersed over a geographical area. They used set of mobility traces that

were recorded at a fine granularity over 25 days of taxis traveling in San Francisco. Taxis would connect to the closest EDC from their location and for each connection the EDCs computing capacity would be reduced by one. Their goal was to accurately predict the loads that would occur in these EDCs. To that end, they used a *Vector Autoregressive Model*. This model is able to predict a signal (here: EDC load) using a linear combination of the signals' own lagged values and the lagged values of the other model variables (here: the EDCs neighbors) and an error term. The researchers showed that this model outperforms the state-of-the-art location-unaware prediction method by 4.3% and achieved an average accuracy of up to 93%.

3.7 Assignment in Fog Environments

In a recent publication, Mohan et al. [62] assign tasks within a Fog environment. It consists of Edge devices, such as desktops, laptops, nano data centers, tables and smartphones and Fog devices, such as routers and switches, with more processing power than Edge devices. Devices can execute tasks, have a certain processing power and are connected via network links. Tasks have a certain processing requirement and may depend on each other. They need to be assigned to devices such that if two jobs depend on each other, the two devices they are assigned to need to be connected via a network link. The two objective functions they used were network costs and processing costs (MOP) and used custom heuristics (*Network Only Cost Assignment*, *Least Processing Cost First Assignment*) to find an solution.

Velasquez et al. [81] also consider a Fog environment and describe an abstract architecture for it, which has to be capable of assigning both tasks (called *services* in their publication) and users to a set of PMs. They suggest to minimize the cost functions (1) *hop count between users and the services* they request and (2) *hop count between communicating PMs*. Their envisioned algorithm can be run multiple times and the assignment created in a previous iteration can be compared to an assignment in the following one. The researchers then suggest to also take into account (3) the *amount of service migrations*, that have to be done, when deriving a new assignment, which should also be minimal. They hint at ILP as one of the methods to find a solution to (1)-(3).

Jia et al. [42] study cloudlet placement and mobile user-to-cloudlet allocation in wireless metropolitan area networks. Users submit computation tasks to a cloudlet they are assigned to and should receive a response from the cloudlet in a timely manner. Each cloudlet is modeled as a queue with a certain maximum task servicing capability. If too many users submit a task, it will become overcrowded. In that case, the cloudlet can forward tasks to the (remote) cloud. The objective is to find an assignment of cloudlets among access points (APs) and users to cloudlets such that the average wait time for offloaded user requests is minimal. Two heuristic algorithms are proposed: Heaviest-AP First and Density-Based Clustering. The first algorithm assigns users to the closest cloudlet and then tries to place cloudlets to those APs where the highest number of workloads arrive. The second algorithm is a refined version of the first, that initially

assigns the cloudlets based on the maximum number of requests that would arrive if all users in its vicinity connected to it. In a second step, it spreads the users among the APs according to a second metric, which causes the user assignment to be spread more evenly.

CHAPTER 4

Assignment in Fog Environment Using BBO

For illustrative purposes, consider an instance of a simplified version of the assignment problem shown in Figure 4.1. Users that are in the vicinity of a base station, as shown by the grey circle, can connect to its PM. Some PMs are connected with each other, and the cost of using a connection may vary. Users need certain VMs, because they inhabit some of the user's applications (apps). The needs for this particular instance are shown in Table 4.1.

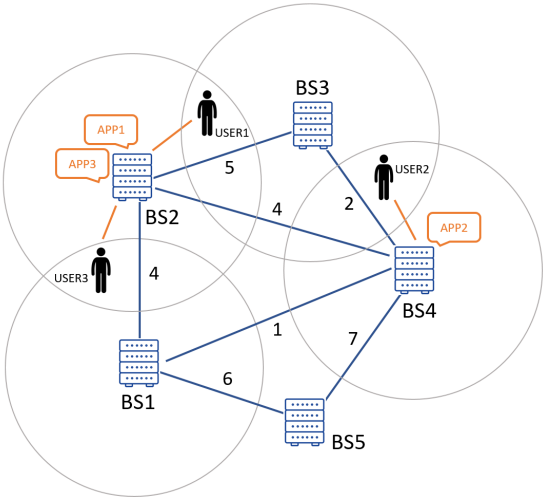


Figure 4.1: A simplified problem instance of the User-VM-assignment problem. The found solution is printed in orange.

Table 4.1: Example for user needs for certain apps.

User	APP1	APP2	APP3
USER1	0	0	3
USER2	1	2	0
USER3	4	0	1

Let the cost of a solution be determined only by how close a user is assigned to the VMs he needs, scaled by how much he needs them. That is, if a user is assigned to the same PM as an app it needs, the cost is zero. Otherwise, the cost is the distance to the app times the amount the app is needed. The cost for the drawn assignment shown in Figure 4.1 is 4, because USER1 and USER3 have all their apps on their server (BS2) and USER2 (assigned to BS4), needs to use the connection between BS4 and BS2 to reach one of its apps and the cost for that connection is 4.

In the following, we will explain a more realistic setting that we chose that includes more constraints (Section 4.1) and more cost functions (Section 4.2).

4.1 Modeling the Fog Environment

This section will explain how the Fog Environment was modeled and the rationale behind the assumptions that were used. The problem we are trying to solve is an assignment problem that is subject to feasibility constraints. We want to find an assignment of both users and virtual machines (VMs) to physical machines (PMs) such that the assignment is feasible. Further, the assignment should be minimal with regards to certain costs, which we will detail in Section 4.2 below. Throughout this chapter we will use the following conventions: There are $1...i...N$ VMs and $1...u...U$ users, that have to be assigned to $1...j...M$ PMs. VMs and PMs have $1..d..D$ dimensions.

4.1.1 Physical Machines

PMs have certain capacities for each of their dimensions (such as CPU, memory, network bandwidth). We will only consider scenarios where the number of dimensions is at least two. For the sake of simplicity, we will assume that all capacities have been normalized to one. All PMs were identically constructed and therefore have the same capacities.

PMs need electricity. Power demand is associated with the current workload of a PM, so we will use CPU utilization to calculate how much electricity it consumes. Let D_j^{CPU} be the dimension of a PM that represents CPU utilization. We will calculate power consumption of PM j using

$$Power(j) = P_j^{idle} + (P_j^{busy} - P_j^{idle}) \cdot D_j^{CPU} \tag{4.1}$$

where P_j^{idle} and P_j^{busy} represent a PMs power consumption at 0% and 100% load, respectively. A PM that has no VMs assigned, does not consume any power.

PMs are connected with each other via network links that have a certain delay. This can be modeled with a simple, weighted graph, in which the vertices and edges represent PMs and their connections, respectively. The delay of two connected machines are the weights of the edges. Not every pair of PMs is necessarily connected with each other, but every PM is reachable from any other PM, via one or more hops. The distance between two PMs is given by the shortest path in the weighted graph.

PMs are known to become unreachable from time to time. This is modeled by assigning each PM a value $fail(j) \in (0..1)$ that represents its failure probability. The failure probability of a PM j can also be described as $1 - Availability(j)$.

4.1.2 Virtual Machines and Users

VMs have certain resource demands for each of their dimension (such as CPU utilization or memory consumption). $load(i, d) \in [0..1]$ resource demand of VM i for the dimension d. No VM has a resource demand that exceeds the capacity limits of any of the PMs. VMs are assumed to be location independent and can therefore be assigned to any PM.

Users, however, can only be assigned to a subset of PMs. This subset may be different for every user, but every user can connect to at least one PM. Users make a certain amount of requests in a time interval to specific VMs and therefore need these VMs. Let $needs(u, i) \in [0..\infty)$ be the matrix that encodes the needs of user u for VM i. Further, let $needs\_bool(u, i)$ be the matrix that results defined as

$$needs\_bool(u,i) = \begin{cases} 1 & \text{if } needs(u,i) \geq 1 \\ 0 & \text{otherwise} \end{cases} \tag{4.2}$$

4.1.3 Solution Encoding and Feasibility

An assignment $\mathbf{x}$ of users and VMs to PMs can be encoded as

$$\mathbf{x} = (\beta_1, \beta_2, ..., \beta_U, \alpha_1, \alpha_2, ..., \alpha_N). \tag{4.3}$$

α_i is a function that returns the PM ID to which VM i is currently assigned, β_u returns the PM ID to which user u is currently assigned.

We define the indicator function $a(i, j)$ and $b(i, j)$ for a specific assignment $\mathbf{x}$ as follows:

$$a(i,j) = \begin{cases} 1 & \text{if VM } i \text{ is assigned to PM } j \\ 0 & \text{otherwise} \end{cases} \tag{4.4}$$

$$b(u,j) = \begin{cases} 1 & \text{if user } u \text{ is assigned to PM } j \\ 0 & \text{otherwise} \end{cases} \tag{4.5}$$

A solution of the assignment problem is feasible, if the following two sets of criteria are fulfilled:

- No PM is used more than it is physically possible. In other words, the sum of resource demands of all the VMs that are assigned to a single PM, does not exceed this PMs' capacities in any dimension.

$$\sum_{i=1}^{N} \sum_{d=1}^{D} load(i,d) \cdot a(i,j) \leq 1 \;\; \forall j = 1..M \tag{4.6}$$

- All the users are assigned to a PM from their set of possible PMs. Let $poss(u)$ denote that set for user u. It has to hold, that

$$b(u,j) = 1 \implies j \in poss(u) \;\; \forall u = 1..U. \tag{4.7}$$

4.2 Cost functions

We want to find an assignment of both users and virtual machines to physical machines, such that the assignment is not only feasible, but also takes the following properties into account:

- **User distance** The distance in terms of latency between a user and the VMs it needs is minimal. In general, the more a user needs a VM, the closer it should be assigned to it. Ideally, all the VMs a user needs are assigned to the same PM as the user himself.
- **Power consumption** Physical machines exhibit a power consumption that is related to their current workload. The fewer machines are used in an assignment, the more power is saved.
- **Resource waste** VMs may have a high demand of one resource (dimension), but a low demand in others. In an ideal assignment, all dimensions of a PM are used equally.
- **Failure probability** Reliable PMs, that is, PM with a high availability guarantee, should be preferred over unreliable ones with a low availability guarantee.
- **Reachability** Network links are also a potential source of failure. To ensure that a user is able to reach the VMs he needs, certain VM locations should be preferred over others, namely PMs that are reachable via multiple paths. These paths should be edge-disjoint, meaning that they do not have any network links in common.

- **User evenness** Each PM should serve approximately the same number of users to ensure an appropriate service quality.

These properties have been formulated as cost-functions $f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_6(\mathbf{x})$ below, which should all be minimized, ideally simultaneously. Before we describe them, we introduce some additional notation.

Auxiliary functions

For a specific assignment c, we define the following auxiliary functions:

- $dist(u, i) \in [0..\infty)$ be the distance between the current assignment of user u and the current assignment of VM i. The distance is 0, if user and VM are assigned to the same PM. If they are assigned to different PMs, we use the shortest path in the graph. Since there are only edges with positive weights and therefore no negative cost cycles, the shortest paths can be calculated using Dijkstra's Algorithm [28].
- $edpaths(u, i) \in [1..\infty)$ is the number of edge-disjoint paths between the current assignment of user u and the current assignment of VM i. The number of edge-disjoint paths between a pair of vertices in an undirected, weighted graph can be determined by setting all edge weights to unit weight and then determining the maximum flow between those vertices.
- $p(j) \in [0..1]$ is the fraction of users that are currently assigned to PM j. If every user is assigned to a different PM, then $p(j) = p(j') \; \forall j, j' \in M$ [74].

Cost functions

User distance

$$f_1(\mathbf{x}) = \sum_{i=1}^{N} \sum_{u=1}^{U} dist(u, i) \cdot needs(u, i) \tag{4.8}$$

Power consumption

$$f_2(\mathbf{x}) = \sum_{i=1}^{N} \sum_{j=1}^{M} Power(load(i, 1)) \cdot a(i, j) \tag{4.9}$$

Resource waste

$$f_3(\mathbf{x}) = \sum_{i=1}^{N} \sum_{d=1}^{D} 1 - \sum_{j=1}^{M} load(i, d) \cdot a(i, j) \tag{4.10}$$

Failure probability

$$f_4(\mathbf{x}) = \sum_{j=1}^{M} [fail(j) \cdot \sum_{i=1}^{N} x(i, j)] \tag{4.11}$$

Reachability

$$f_5(\mathbf{x}) = \sum_{i=1}^{N} \sum_{u=1}^{U} \frac{dist(u,i) \cdot needs\_bool(u,i)}{edpaths(u,i)} \tag{4.12}$$

User evenness

$$f_6(\mathbf{x}) = \sum_{j=1}^{M} p(j)^2 \tag{4.13}$$

Minimizing user distance (f_1) assures that the delay user experience will be minimal. The more frequently a user makes a request to a particular VM, the closer it should be assigned. Therefore, the distance is scaled by the needs. Minimizing resource waste (f_3) assures that there are no spaces when VMs are assigned to PMs (multidimensional bin packing). This ensures that capacities of PMs are not wasted and PMs are used efficiently. The cost function for failure probability (f_4) ensures that reliable PMs are favored over unreliable ones. Reachability (f_5) is minimal if the user and the VMs they need are placed on the same PM. Otherwise, it is proportional to the number of paths that exist from the users to their VMs. Finally, user evenness (f_6) is minimal, if all users are assigned to different PMs and ensures that a single PM does not get overcrowded. Equation 4.13 is also called Simpson Index.

4.2.1 Pareto dominance

In Section 4.2 we stated, that we seek solutions that minimize several cost functions *simultaneously*. However, there are situations, where it is not clear which solution is preferred over which. Imagine the following scenario: Let $\mathbf{x}$ be a feasible solution and let

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), ..., f_q(\mathbf{x})] \tag{4.14}$$

be its corresponding vector of cost functions. Let $\mathbf{x}'$ be another feasible solution and $\mathbf{y}'$ be its costs. Which solution is preferred if $\mathbf{y}(i) > \mathbf{y}'(i)$ but $\mathbf{y}(j) < \mathbf{y}'(j)$ for $i \neq j$ and $1 \leq i, j \leq q$?

For this reason, we use the concept of *Pareto dominance*. We say $\mathbf{x}$ dominates $\mathbf{x}'$, denoted as $\mathbf{x} \succ \mathbf{x}'$, if $\mathbf{f}(\mathbf{x})$ is less than or equal to $\mathbf{f}(\mathbf{x}')$ in every objective function and strictly less in at least one objective function. We say $\mathbf{x}$ and $\mathbf{x}'$ are non-comparable, denoted as $\mathbf{x} \sim \mathbf{x}'$, if neither $\mathbf{x}$ dominates $\mathbf{x}'$ nor $\mathbf{x}'$ dominates $\mathbf{x}$ [65]. So in the above scenario, both solutions are *non-comparable*. Algorithm 4.1 shows the pseudo-code for comparing two

solutions.

Algorithm 4.1: Pareto Dominance

Data: $\mathbf{y}, \mathbf{y}'$, two objective function arrays, that correspond to the two solutions $\mathbf{x}, \mathbf{x}'$

Result: Returns either *true* if $\mathbf{x} \succ \mathbf{x}'$ or *false* otherwise.

```
1 flag ← false;
2 for i ∈ {1..q} do
3 |   if y[i] > y'[i] then return false ;
4 |   if y[i] < y'[i] then flag ← true ;
5 end
6 return flag
```

4.3 Problem-Specific Algorithm Adaptations

This section details how certain algorithm steps were adapted, because, first, we are dealing with a *discrete*, rather then a continuous problem and, second we have *problem-specific constraints*.

4.3.1 Initial Solutions

In Section 2.3.1, we discussed the BBO algorithm, and how it derives a generation of solutions from the previous ones. The initial generation, however, must be provided to the algorithm. We decided to generate these initial solutions greedily. More precisely, we use a greedy heuristic that tries to minimize the distance between users and their VMs, which minimizes objective function f_1 (Equation 4.8), while trying to be not too wasteful with PM resources (functions f_2 and f_3, Equations 4.9 and 4.10).

The reason for using greedy initial solutions minimizing f_1 is the following. If our problem definition was formalized differently and we were only considering f_1 as our single objective function, our user/VM assignment problem would be similar to the famous Quadratic Assignment Problem (QAP) introduced in 1957 by Koopmans and Beckmann [48]. This problem is NP-hard and has been well studied in the past. Strategies for solving QAP include using Genetic Algorithms and BBO and both of which generate their initial population either randomly (e.g. [80, 41, 54]) or using a greedy strategy (e.g. [20]).

Let $\mathbf{x} = (\beta_1, \beta_2, ..., \beta_U, \alpha_1, \alpha_2, ..., \alpha_N)$ be a solution that we want to create as defined in Section 4.1.3. The first part of the solution (the user assignment), is created by randomly assigning users to one PM. More precisely, each β_u in $\mathbf{x}$ is derived by selecting a PM

uniformly at random from $poss(u)$ with $1 \leq u \leq U$. A placement for VMs, the second part of $\mathbf{x}$, is found according to Algorithm 4.2.

Algorithm 4.2: Greedy Assignment of VMs

1. Let $\mathbf{x}$ be a partially filled solution, that has users assigned but not VMs
2. Sort all users by their ID
3. For each user u
   a) If all VMs have been assigned in $\mathbf{x}$, terminate.
   b) Put all the VMs user u needs into a list L and sort L by the needs of u decreasingly. The VM that is needed the most is therefore at the top of L, the VM that is needed the least is at the bottom of L. VMs that are not needed by u are not present in L.
   c) Remove all VMs from L that were already assigned in $\mathbf{x}$.
   d) For each VM v in L
      i. If v can be assigned to the same PM as u, assign it.
      ii. Otherwise, iterate over all other PMs in increasing order of distance. Assign v to the first PM, that is capable of holding it.

This procedure generates different solutions each time it is executed, because it contains a random component and can therefore be run several times to create the initial generation. It is also guaranteed that every solution is feasible.

4.3.2 Mutation of Solutions

In the original publication of BBO [73], the input variables to the objective function are continuous. Each of these continuous variables has a certain allowed range of values that it can take on (domain of definition). When a new, mutated solution is generated, the value of a single variable is replaced by new value randomly chosen from that variables' domain of definition. We adapt this procedure, to make it applicable to this specific discrete problem.

Let $\mathbf{x}$ be a feasible solution that we want to alter in one variable, such that a new, different solution is created and let v_i be that variable. We distinguish two cases: If $i \leq U$, we replace i by choosing a new PM by choosing a random VM from $poss(i)$, similar to the initial solution generation from Section 4.3.1. If $i > U$, we choose a random PM from the whole set of possible PMs. This may yield an infeasible solution, since PMs have a limited capacity, and therefore we immediately repair the solution as described in Section 4.3.3.

4.3.3 Repair of Solutions

Repairing solutions is sometimes necessary, because the migration and mutation step of the BBO algorithm may yield infeasible solutions, as we have already discussed in Section 2.3.5.

Let $\mathbf{x}$ be an infeasible solution and let I be the set of indices of the variables that contains (1) the users, that are infeasibly assigned and (2) the PMs, that are over their capacity limits (see also Section 4.1.3). For each $i \in I$ with $i \leq U$, do the mutation procedure from Section 4.3.2. For each $i \in I$ with $i > U$, pick a random VM v from the overloaded PM i. Then apply the last step from Algorithm 4.2 again: Iterate over all other PMs in increasing order of distance. Assign v to the first PM, that is capable of holding it.

After this procedure, $\mathbf{x}$ is guaranteed to be feasible.

CHAPTER 5

Evaluation

Chapter 2 was concerned with creating a model for the fog environment and the overall strategy for solving the assignment problem. In this Chapter we build upon these models and strategies and discuss how the concrete parameters were chosen for both of them. We further give two reference strategies to evaluate against, briefly touch on some implementation specific issues and finally present the results of this work. The results section contains several parts: Section 5.4.1 contains the results for monoobjective optimization, while Section 5.4.2 contains the general results for multiobjective optimization. Section 5.4.5 describes another set of experiments that used a different underlying topology. Section 5.5 discusses limitations and opportunities for future work. The conclusion is given in Section 5.6.

5.1 Experiment Setup

The environment of all test cases is generated randomly, given a number of parameters. The behavior of the BBO algorithm is also driven by some parameters that have to be provided at the start of the simulations. A summary over both environment and BBO parameters alongside an explanation is given in Table 5.1.

5.1.1 BBO Parameters

The parameters `ngen` and `popsize` control the number of solutions that the BBO algorithm produces and tests. Both parameters together are the main drivers of the runtime of the algorithm. The shape of the immigration and emigration functions, `tfunc`, could either be set to *linear* or *sinusoidal*. We set `tfunc` to *sinusoidal* in all experiments, except for the simulations where its impact on the solution quality was assessed (Table 5.7). The mutation probability, `pmutate`, determines the likelihood of a single individual to mutate in the BBO algorithm after the migration step. It was set to 0.01 in all experiments,

Table 5.1: Parameters for execution of the BBO algorithm and the test environment.

Short Name	Explanation
BBO parameters	
`ngen`	Maximum number of generations.
`popsize`	Population size of every generation.
`tfunc`	Type of migration functions determines the shape of both μ and λ.
`pmutate`	Mutation probability of an individual.
`keep`	The number of individuals to preserve from one generation to the next (elites).
Environment parameters	
`nvms`	Number of Virtual Machines (VMs).
`npms`	Number of Physical Machines (PMs).
`nusers`	Number of users.
`ndim`	The number of dimensions of both VMs and PMs, such as CPU, memory or network bandwidth usage.
`nuserconnections`	The number of PMs which a user is able to connect to.
`pmavail`	Interval for the range of the availabilities for a PM.
`pmpower`	Power consumption in Watts of a PM. All PMs are assumed to be constructed identically.

which was the original authors suggestion, unless in the ones that evaluate its impact. Finally, the number of solutions to carry over from one iteration of the algorithm to the next, `keep`, was set to twice the number of cost functions, that were being evaluated. In other words, `keep`=2 in the monoobjective experiments and `keep`=12, where all of the cost functions were used. This strategy guarantees that (at least) the best two solutions of each cost function are preserved from one generation to the next.

5.1.2 Environment Parameters

`nvms`, `npms` and `nusers` determine the size of the solution space. Table 1.1 lists the maximum number of solutions the solution space can have, given those three parameters (but ignoring other constraints). Changing some or all of those parameters has a significant influence on how long it takes to evaluate each of the six cost functions. Further, `npms` determines how many nodes the underlying network infrastructure graph has that connects all the PMs.

`ndim`, `nuserconnections`, `pmavail` and `pmpower` were fixed parameters in all experiments. The number of dimensions of VMs and PMs, `ndim`, was set to 2. This means both VMs

Table 5.2: Physical machine power consumption in Watts by percentage of utilization [17].

Physical machine	0%	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
HP ProLiant G4	86	89.4	92.6	96	99.5	102	106	108	112	114	117

and PMs are considered to have only two limiting capacities, CPU and memory. We could have set it to a larger value, but the BBO algorithm only uses `ndim` when it evaluates cost function f_3 (Resource waste) and when the feasibility of a solution is checked. Increasing `ndim` to 3 or more would unnecessarily slow down cost function evaluation, and not necessarily yield any new information regarding the efficacy of our algorithmic approach.

The number of EDC a user can be assigned to, `nuserconnections`, was set to 3. This means, that there are always exactly three choices to assign a user. Figure 1.1a shows a real-world example of the geographical distribution of cell towers in and around the center of Vienna. In this Figure, it is obvious that users could connect to more than three base stations. However, we chose a conservative estimate, that also holds true for less densely populated areas.

The availability interval, `pmavail`, of all PMs was fixed to $[0.99, 0.99999]$. At the start of a simulation, the availability of each PM is set to a value chosen uniformly at random from this interval. For example, an availability of 99% equals a downtime of 7.20 h/month, while an availability of 99.999% equals a downtime of 25.9 sec/month. Availability guarantees of cloud providers often fall in this range[1].

Further, the power consumption parameter, `pmpower`, was fixed for all physical machines. Their values are shown in Table 5.2 by percentage of server utilization. They stem from a *HP ProLiant G4* server (Intel Xeon 3040, 2 cores at 1860 MHz, 4 GB of memory) and are the result of the SPECpower benchmark [17]. They have been used in the past by other researchers to evaluate cloud data center scheduling policies [25].

5.1.3 Environment Generation

Based on the parameters discussed in the previous Section, several other entities are generated to arrive at the full experiment setup. Some of the generation processes are based on commonly used, statistical distributions. Their notation, mean and variance are given in Table 5.3.

Table 5.3: Notation for statistical distributions.

Symbol	Distribution Name	Mean	Variance
$\mathcal{N}(\mu, \sigma^2)$	Normal Distribution	μ	σ^2
$\mathcal{P}(\lambda)$	Poisson Distribution	λ	λ
$\mathcal{U}\{a, b\}$	Uniform Distribution, discrete	$\frac{1}{2}(a+b)$	$\frac{1}{12}(b-a)^2$
$\mathcal{U}(a, b)$	Uniform Distribution, continuous	$\frac{1}{2}(a+b)$	$\frac{1}{12}(b-a)^2$

[1] Amazon EC2 SLA - `https://aws.amazon.com/ec2/sla/`

VM generation VMs exhibit a certain load. We will assume that the distribution underlying all VMs is identical. Further, we assume that all components of each VM are drawn from the same distribution. This assumption is based on previous work by Zheng et al. [91]. For that purpose, we chose a Normal distribution of $\mathcal{N}(\mu = 0.15, \sigma^2 = 0.05)$ (measured in fraction of PM capacity). However, other modeling approaches do exist: Jin et al. [43] modeled VMs using 4 components, 1 of which is larger (between 30-40% of the capacity of a PM) than the other 3 which are small (between 5-10%). Xu et al. [86] uniformly chose CPU demand (measured in GHz) from the set {0.25 0.5 1 1.5 2 2.5 3} and memory demand (measured in GB) from the set {0.25 0.5 1 1.5 2 2.5 3 4}. Ruan et al. [70] set all VMs to the same size.

User generation Users connect to PMs and then make requests to VMs. Users have a certain location and as a result are only capable of connecting to a subset of all PMs of size `nuserconnections`. These PMs are determined by choosing `nuserconnections` PMs uniformly at random from the set of all PMs. Further, users are assumed to make a number of requests per time interval to a VM. Modeling this count data, we used a Poisson distribution $\mathcal{P}(\lambda = 10)$.

PM failure generation Each PMs individual failure probability is generated by drawing from $\mathcal{U}(a, b)$, where a, b are the bounds of `pmavail`.

Finally, **PM network generation** has to take place. As we already explained in Section 1.1, we assume that we have a virtualized bandwidth guaranteed network at our disposal, that provides enough bandwidth for all of our transmission purposes. Therefore, our main concern is modeling the appropriate connections (network layout) between PMs and the appropriate delays (measured in milliseconds). Both are represented by a weighted graph $G = (V, E)$, which is generated in multiple steps. First, a complete graph K_n is generated by drawing the all of its edge weights from $\mathcal{U}(a = 10, b = 20)$. This assures that the graph is fully connected as it contains $\frac{n \cdot (n-1)}{2}$ edges, where $n = |V|$. In this graph, we can reach any PM from any other PM in one hop. Next, a minimum spanning tree is derived, which is done via MATLABs `minspantree()` function, that implements Prim's algorithm[2]. This tree contains $n - 1$ edges. In this graph, we still can reach any PM from any other PM, however, there is exactly one path to do so. For our evaluation, we desire something in between a fully connected graph and a tree, and therefore calculate $\overline{n} = \frac{n \cdot (n-1)}{4}$. While $|E| < \overline{n}$, we select a pair of non-adjacent vertices and add an edge between them. The edge weight again is drawn from $\mathcal{U}(a = 10, b = 20)$.

This results in a PM connection architecture, where every PM is connected to half of the other PMs on average, and the whole graph is guaranteed to be connected. In Section 5.4.5, we will use a different graph layout, that more accurately reflects the graph topology of the Internet, but all edge weights we will be drawn from the same uniform distribution.

[2]MATLAB reference: `https://de.mathworks.com/help/matlab/ref/graph.minspantree.html`

5.2 Comparison Algorithms

Our approach was evaluated against two reference strategies; a greedy strategy (*Greedy*) and a genetic algorithm (*GA*).

5.2.1 Greedy Strategy

This algorithm finds a list of initial solutions and iteratively tries to improve them. Initial solutions are found the same way as for the BBO algorithm (see Section 4.3.1). Let S denote the list of solutions and $|S| = q$ is the number of objective functions. In every iteration, all solutions $s \in S$ are investigated by exploring a random subset of their neighbors. This is done by first mutating 10% of the variables in s and then repairing s, in case it was infeasible. Mutation and repair procedures are explained in Sections 4.3.2 and 4.3.3, respectively.

After mutation and repair, the algorithm determines if the new solutions are better than the existing ones. Let S' be the new list of solutions that were created based on S. Iterate over S' and let j be the iteration variable. If solution $S'[j]$ dominates $S[j]$, set $S[j] \leftarrow S'[j]$. Domination is assessed by Algorithm 4.1.

The Greedy procedure stops after a predefined number of iterations have taken place, which was set to 10000 in all experiments.

5.2.2 Genetic Algorithm

GA [61] was created to mimic the *survival of the fittest* process from nature, where the best adapted species survive the longest and are able to reproduce. First, a set of initial solutions, also called a generation, is created (see Section 4.3.1). Then, these solutions are mutated, repaired and ranked. Solutions with a higher quality are ranked higher, while poor solutions will be have a lower rank. After that, solutions are recombined with each other. Being selected for recombination depends on the rank of a solution. Recombination yields a new offspring generation. This new generation is again ranked and recombined. As a result, GA iteratively improves a list of solutions from one generation to the next. Ranking, mutation and repair functions are identical to the ones we used for BBO (see Sections 2.3.2, 4.3.2 and 4.3.3, respectively).

Recombination

Recombination is called *crossing-over* and works as follows. Several methods for crossing-over exist. We chose *uniform crossover*, which selects variables from both solutions uniformly. It is implemented as follows. We are given two solutions, as the result of the selection process explained below, $\mathbf{x_A}$ and $\mathbf{x_B}$ from the existing generation. Let p be the number of variables they contain (the *length* of those solutions). We create a boolean $0/1$ random vector u that has dimension p. The components $u(i)$ of the vector have the same probability of occurrence, $P(u(i) = 0) = P(u(i) = 1) = 0.5$.

The new, recombined solution $\mathbf{x}_{AB}$ is derived by

$$\mathbf{x}_{AB}(i) = \begin{cases} \mathbf{x}_A(i) & \text{if } u(i) = 0 \\ \mathbf{x}_B(i) & \text{if } u(i) = 1 \end{cases} \tag{5.1}$$

for $1 \leq i \leq p$.

Ranking, Mutation and Selection

After recombination, solutions are mutated and repaired and then ranked based on their fitness functions. For ranking, we implemented the same adaptation as for BBO, to being able to rank according to multiple objective functions. After ranking, the top 25% of solutions are mutated and repaired and replace the bottom 25% of solutions. Finally, after another ranking of solutions, the elites of the previous generation, replace the worst solutions. Then, the next iteration of the algorithm begins.

The selection of two solutions for crossing over is based on their ranking. The lower the costs, the higher the ranking and the more likely it is for a solution to be chosen. This is called *fitness proportionate selection* [61, p. 124f] and the probability p_i for a solution i to be chosen is

$$p_i = \frac{\mu_i}{\sum_{j=1}^{S_{max}} \mu_j}, \tag{5.2}$$

where μ_i is

$$\mu_i = \frac{S_{max} + 1 - r_i}{S_{max} + 1}, \tag{5.3}$$

r_i is the rank of solution i and S_{max} is the number of solutions in the generation. This is virtually identical to the Roulette-wheel selection of BBO and Equation 2.7.

5.3 Implementation and Setup Details

All algorithms were implemented in MATLAB™ [67]. Several scripts from the original publication by Simon [14] and a follow-up publication by Khademi et al. [8] are published online. These scripts were used as a template and were one of the reasons why MATLAB was chosen as the simulation environment. The automatic generation of test instances and their components such as the structure of the underlying network graph, the sizes of VMs, the location of the users and all other necessary components were also implemented in MATLAB. The fact that a variety of graph statistics tool (such as graph centrality measures) and graph algorithms (such as `minspantree()`) are readily available in MATLAB, was another reason why we chose this simulation environment.

Some auxiliary Python [15] scripts were written to output the results into table format. Overall, there are 33 MATLAB files, totaling 3115 lines, and 4 Python files, totaling 285

lines[3]. All simulations were carried out on a HP Notebook (Intel Core i5 with 2 cores running at 2.30 GHz each; 16 GB RAM).

5.4 Results and Discussion

Ideally, we would like to have exact solutions to the problem instances for comparison, however, the large solution space does not allow it to compute them easily (see also Table 1.1). In order to test the *correctness* of our BBO implementation, very small instances were created (`nvms`, `nusers`, `npms` ≤ 5) and exact solutions were derived for those via enumeration. The BBO algorithm was able to compute the same (or slightly worse) results, giving us confidence in the correctness of the implementation (results not shown). The genetic algorithm was able to do the same.

5.4.1 Monoobjective Results

As a first set of simulations, we minimized all the objective functions individually. This was in part done to get some intuition for which kind of solution values we can expect from the chosen problem instances. Plots of results using the largest tested instance are depicted in Figure 5.1 which shows the convergence behavior of the cost functions (y-axis) by number of generations (x-axis); the full results are summarized in Table 5.4.

[3]The tool *CLOC* was used to determine these metrics and is available at `https://github.com/AlDanial/cloc`.

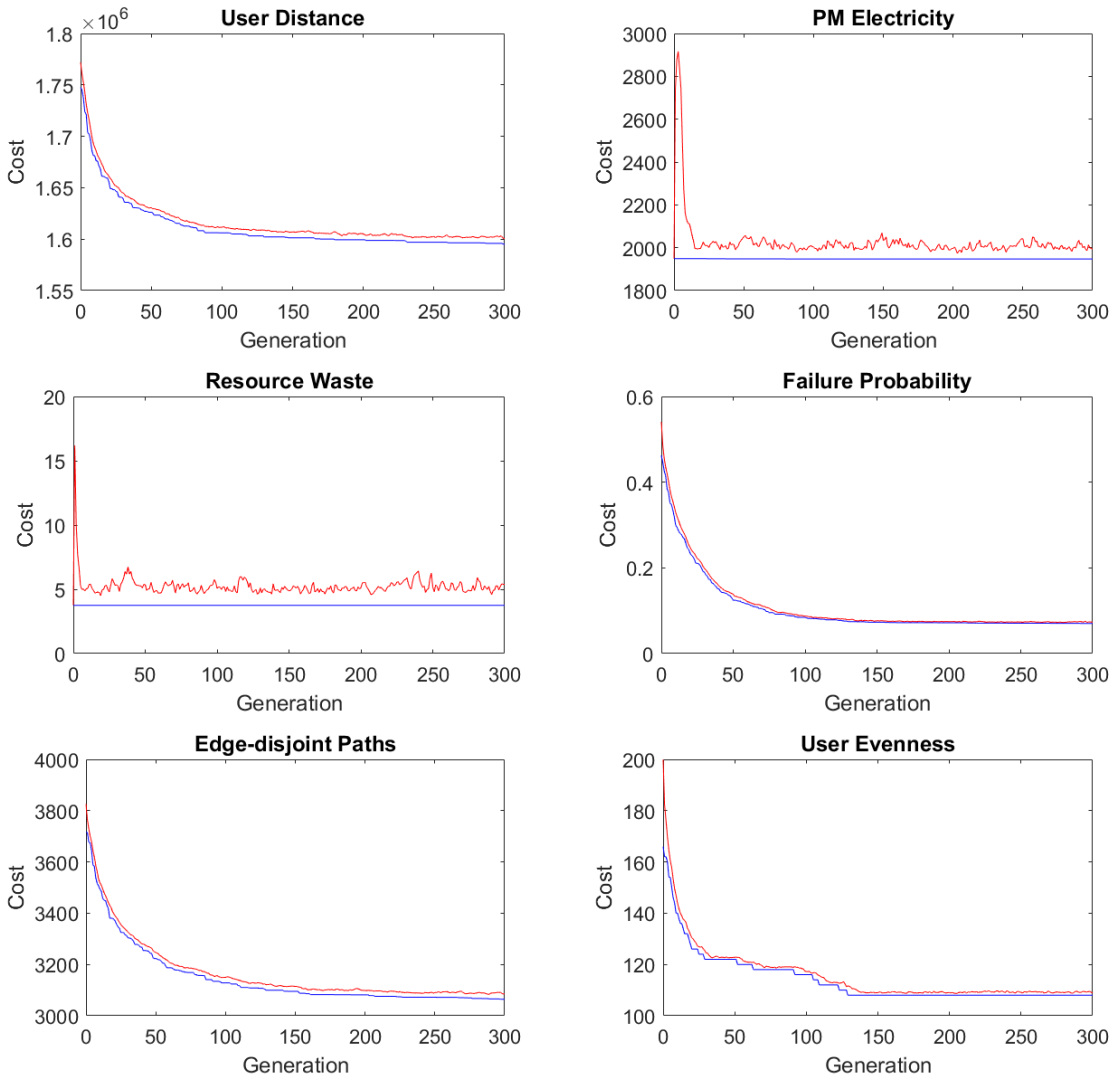


Figure 5.1: Results of the BBO algorithm minimizing the six objective functions individually. Figure depicts the scenario where `nvms`=100, `npms`=100 and `nusers`=100 and BBO was started with a generation size `ngen`=50 (test cases 19-24 in Table 5.4). Average of the cost function of a generation depicted in red, minimum of a generation in blue.

In Figure 5.1, the minimum cost of every generation is drawn in blue, the average cost is drawn in red. As can be seen clearly, the minima are always monotonically decreasing. This is because the best two solutions for each cost function are always chosen as elites and therefore preserved from one generation to the next. The average costs (red line), are also decreasing from one generation to the next for some of the cost functions (*user distance*, *failure probability*, *edge-disjoint paths* and *user evenness*). For other cost functions (*electricity costs*, *resource waste*), the minimum stays the same while the average oscillates and does not decrease from one generation to the next. This could be due to the fact

Table 5.4: Results of minimizing each of the six objective functions individually.

No.	nvms	npms	nusers	popsize	ngen	Costs	Best Greedy	Best BBO	Best GA	$\frac{BBO}{Greedy}$	$\frac{BBO}{GA}$
1	25	25	25	50	300	f_1	77346.0000	72636.0000	73955.0000	0.94	0.98
2	25	25	25	50	300	f_2	570.9187	570.8945	570.8930	1	1
3	25	25	25	50	300	f_3	1.7899	1.7899	1.7899	1	1
4	25	25	25	50	300	f_4	0.0417	0.0417	0.0433	1	0.96
5	25	25	25	50	300	f_5	657.0193	584.7590	600.2455	0.89	0.97
6	25	25	25	50	300	f_6	464.0000	464.0000	464.0000	1	1
7	50	50	50	50	300	f_1	382848.0000	349524.0000	363548.0000	0.91	0.96
8	50	50	50	50	300	f_2	1029.4583	1029.3620	1029.3620	1	1
9	50	50	50	50	300	f_3	2.3846	2.3846	2.3846	1	1
10	50	50	50	50	300	f_4	0.0633	0.0418	0.0417	0.66	1
11	50	50	50	50	300	f_5	1554.8663	1403.6661	1426.6085	0.9	0.98
12	50	50	50	50	300	f_6	232.0000	224.0000	224.0000	0.97	1
13	75	75	75	50	300	f_1	913635.0000	850194.0000	886814.0000	0.93	0.96
14	75	75	75	50	300	f_2	1491.8871	1490.7479	1490.8703	1	1
15	75	75	75	50	300	f_3	3.2695	3.2695	3.2695	1	1
16	75	75	75	50	300	f_4	0.0984	0.0597	0.0692	0.61	0.86
17	75	75	75	50	300	f_5	2494.9406	2249.4691	2344.5369	0.9	0.96
18	75	75	75	50	300	f_6	168.8889	151.1111	158.2222	0.89	0.96
19	100	100	100	50	300	f_1	1689689.0000	1595252.0000	1631001.0000	0.94	0.98
20	100	100	100	50	300	f_2	1948.7193	1947.5218	1948.3727	1	1
21	100	100	100	50	300	f_3	3.7480	3.7480	3.7480	1	1
22	100	100	100	50	300	f_4	0.1592	0.0702	0.1020	0.44	0.69
23	100	100	100	50	300	f_5	3410.2372	3064.1655	3248.5522	0.9	0.94
24	100	100	100	50	300	f_6	130.0000	108.0000	118.0000	0.83	0.92

BBO = Biogeography-Based Optimization with sinusoidal migration functions, GA = Genetic Algorithm with fitness proportionate selection. The last two columns show the ratio between BBO and the other algorithms, rounded to two decimal places.

that the initial solution used for starting the BBO algorithm was already very close to optimal. Another explanation might be that the BBO procedure, due to its very general approach of searching through the solution space, is not powerful enough to choose good neighboring solutions that are both feasible *and* able to minimize these specific cost functions.

Figure 5.1 also unveils another noteworthy behavior: Functions f_1, f_4, f_5 and f_6 show a exponential function-like shape: During the first iterations, there is a stronger gradient because these initial solutions can be improved a lot. As the generation numbers get higher, the gradient decreases because the found solutions cannot be improved any further. This behavior can be an indicator for the algorithm finding/converging towards an optimum.

Table 5.4 shows the results of the monoobjective simulation runs. In all test cases, BBO was able to outperform the Greedy approach. In 22 out of 24 test cases, BBO was able to find equally good (9/22 test cases) or better (13/22) solutions than the Genetic algorithm. All test cases using the objective functions f_1 (*user distance*) or f_4 (*failure probability*), resulted in better objective function values when solved with BBO, showing that in these scenarios, BBO should be preferred over the other two reference algorithms. The

most improvement of BBO compared to both reference algorithms, was in the biggest instance (`nvms`=100, `npms`=100, `nusers`=100) for cost function f_4 (test case 22), where the found solution was 69% of the cost of the GA solution and 44% of the cost of the Greedy solution.

5.4.2 Multiobjective Results

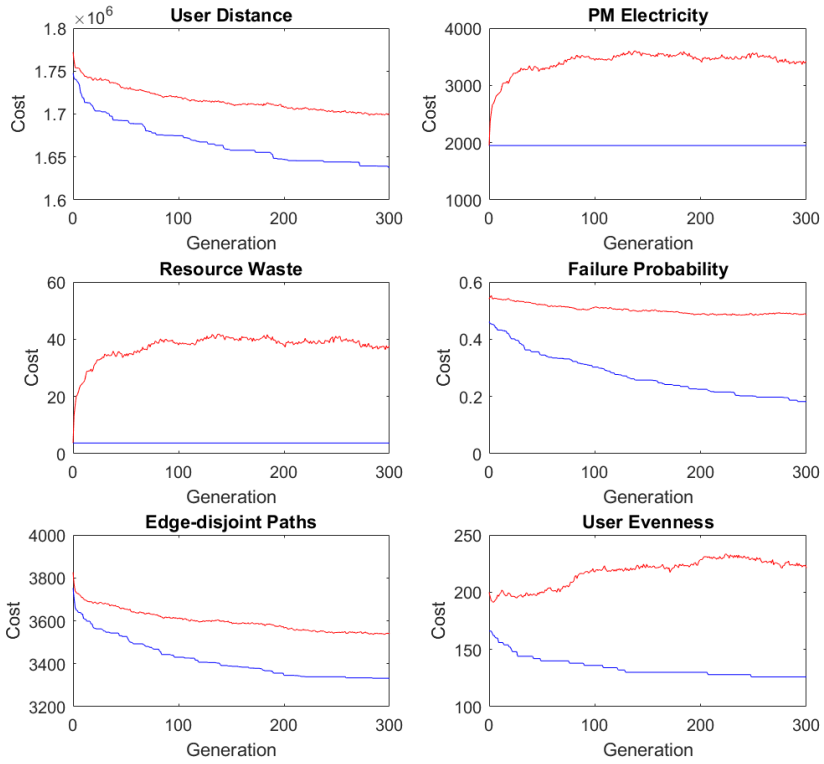


Figure 5.2: Results of the VM-User assignment problem with $nvms = 100$, $npms = 100$ and $nusers = 100$. All objective functions were minimized simultaneously using Biogeography-Based Optimization with $popsize = 50$ and sinusoidal migration functions. Average of the cost function of a generation depicted in red, minimum of a generation in blue.

Figure 5.2 depicts the results for the multiobjective simulation runs. Similar to what we have seen in Section 5.4.1 with monoobjective optimization, minimum and average generation values for some cost functions (f_1, f_4, f_5) decrease, while for other cost functions (f_2, f_3), minimum values stay constant and average values do not decrease. Interestingly, cost functions f_2 and f_3 show an identical shape, yet on a different scale. Intuitively, this makes sense: Solutions that only use a few PMs, use less electricity (f_2) and at the same time are able to pack more VMs on fewer PMs therefore waste less

resources (f_3). Therefore, good solutions for f_2 are also good solutions for f_3.

An issue with multiobjective optimization occurs, when different cost functions require almost opposite solutions to reach their global minimum: The cost functions *user distance* f_1 and *user evenness* f_6 are a good example for this. While f_1 would be minimal is all users were assigned to a *single* PM and all the VMs were assigned to that PM as well (ignoring all other constraints for the moment), f_6 would be minimal is all users were spread evenly among *all* PMs. These two goals conflict with each other, and thus making it also difficult for the BBO procedure to find a reasonably good solution for both targets and to converge towards an optimum suitable for both functions.

The results for the multiobjective simulation are summarized in Table 5.5. We evaluated the BBO algorithm and the two reference algorithms (Greedy and GA) with different instance sizes of the assignment problem. For BBO and GA intermediate results after certain number of generations are shown as well as the time it took to derive them. Since each generation takes approximately the same time to derive, an almost linear relationship between generation size and execution time can been seen.

Both BBO and GA clearly outperform the Greedy approach. The gap between BBO and GA, however is much smaller or non-existent. In terms of objective function values, both approaches are virtually identical for functions f_2 and f_3, most likely due to the issue mentioned above. For objective functions f_1, f_5, f_6, the ratios of the values of the functions stay relatively close to 1 (between 0.980 and 1.107). For objective function f_4, ratios are in the range of 0.895 and 1.416, with GA performing better than BBO in many cases. Still, there is no clear indication of superiority of one approach over the other.

Table 5.6 shows the full set of solutions for the biggest tested instance size, which was also summarized in Table 5.5, test cases 16 and 28 for BBO and GA, respectively. Minimum values are printed in bold. This table tells a more comprehensive story of the quality of solutions produced by each algorithm and shows clearly that there can be huge differences in the values of the individual cost functions, depending on which solution is chosen. A solution might have a small value for cost function f_i, but at the same time a high value for cost function f_j. This is very likely due to the nature of the cost functions and the different goals they are trying to achieve and also a property of pure multiobjective optimization using Pareto dominance. The means of the set of solutions are also shown in Table 5.6. BBO seems to yield better (lower) means of its population, however these differences are small.

Table 5.5: Results of the VM/User assignment problem. Only the best (=smallest) values are shown for the respective fitness functions.

No.	Alg.	nvms	npms	nusers	popsize	ngen	Time [s]	f_1	f_2	f_3	f_4	f_5	f_6
1	Greedy	25	25	25			87	94858	571.2047	1.7899	0.0869	943.6006	656
2	Greedy	50	50	50			171	413432	1029.4583	2.3846	0.3111	1739.2535	384
3	Greedy	75	75	75			290	994339	1491.8871	3.2695	0.2888	2863.4670	229
4	Greedy	100	100	100			431	1769050	1948.7193	3.7480	0.5571	3766.7912	198
5	BBO	25	25	25	50	100	11	75462	570.9305	1.7899	0.0462	612.0697	496
6	BBO	25	25	25	50	200	20	74344	570.9305	1.7899	0.0444	598.0856	496
7	BBO	25	25	25	50	300	30	74032	570.9305	1.7899	0.0441	593.8344	496
8	BBO	50	50	50	50	100	20	376528	1029.4583	2.3846	0.1290	1522.8820	256
9	BBO	50	50	50	50	200	39	369163	1029.4583	2.3846	0.0900	1490.0098	248
10	BBO	50	50	50	50	300	59	365179	1029.4583	2.3846	0.0790	1449.1808	240
11	BBO	75	75	75	50	100	34	932467	1491.8871	3.2695	0.1628	2516.1134	187
12	BBO	75	75	75	50	200	66	913781	1491.6310	3.2695	0.1170	2438.9061	176
13	BBO	75	75	75	50	300	99	909537	1491.6310	3.2695	0.0983	2400.6016	169
14	BBO	100	100	100	50	100	46	1674780	1948.7193	3.7480	0.3024	3430.1291	136
15	BBO	100	100	100	50	200	93	1647191	1948.6540	3.7480	0.2249	3356.3118	130
16	BBO	100	100	100	50	300	140	1637982	1948.6540	3.7480	0.1822	3332.3777	126
17	GA	25	25	25	50	100	11	76545	570.9305	1.7899	0.0516	617.5083	496
18	GA	25	25	25	50	200	21	74487	570.9305	1.7899	0.0446	596.0630	496
19	GA	25	25	25	50	300	29	74084	570.9305	1.7899	0.0433	593.0515	496
20	GA	50	50	50	50	100	21	379604	1029.4583	2.3846	0.1058	1554.7508	248
21	GA	50	50	50	50	200	41	367373	1029.4583	2.3846	0.0651	1497.2110	248
22	GA	50	50	50	50	300	57	362032	1029.4583	2.3846	0.0558	1445.8811	240
23	GA	75	75	75	50	100	33	921074	1491.8871	3.2695	0.1726	2541.4253	169
24	GA	75	75	75	50	200	67	900251	1491.8871	3.2695	0.1244	2452.3447	162
25	GA	75	75	75	50	300	95	890435	1491.8871	3.2695	0.0931	2398.1547	155
26	GA	100	100	100	50	100	48	1693733	1948.7193	3.7480	0.2885	3498.3165	130
27	GA	100	100	100	50	200	93	1668861	1948.7193	3.7480	0.2046	3392.2322	126
28	GA	100	100	100	50	300	132	1649854	1948.7193	3.7480	0.1542	3319.8972	124

BBO = Biogeography-Based Optimization with sinusoidal migration functions, GA = Genetic Algorithm with fitness proportionate selection. `pmutate`=0.01 for both algorithms.

Table 5.6: Full set of solutions from both BBO and GA, with `nvms`=100, `npms`=100 and `nusers`=100. Each solution is minimal for (at least) one of the cost functions f_1 - f_6.

Algorithm	$\mathbf{f_1}$	$\mathbf{f_2}$	$\mathbf{f_3}$	$\mathbf{f_4}$	$\mathbf{f_5}$	$\mathbf{f_6}$
BBO						
	1637982	2997	27.7	0.5383	3416	270.0
	1679233	**1949**	3.7	0.5846	3513	236.0
	1683438	1950	**3.7**	0.5802	3529	234.0
	1770945	3948	49.7	**0.1822**	3732	218.0
	1674186	3946	49.7	0.4925	**3332**	268.0
	1726486	4467	61.7	0.5398	3637	**126.0**
Mean	*1695378*	*3209*	*32.7*	*0.4863*	*3527*	*225.3*
GA						
	1649854	3250	33.7	0.5437	3458	270.0
	1666935	**1949**	3.7	0.5851	3461	238.0
	1667415	1949	**3.7**	0.5851	3464	242.0
	1789088	3865	47.7	**0.1542**	3846	186.0
	1669419	4296	57.7	0.5358	**3320**	282.0
	1733834	4726	67.7	0.5530	3639	**124.0**
Mean	*1696091*	*3339*	*35.7*	*0.4928*	*3531*	*223.7*

5.4.3 Influence of Different BBO Parameters

Table 5.7: Results for different starting parameters for the BBO Algorithm with `nvms`=50, `npms`=50 and `nusers`=50; `p` = mutation probability.

No.	Algorithm	popsize	ngen	Time [s]	f_1	f_2	f_3	f_4	f_5	f_6
1	BBO	50	100	20	376528	1029.46	2.3846	0.1290	1522.88	256
2		50	200	40	369163	1029.46	2.3846	0.0900	1490.01	248
3		50	300	60	365179	1029.46	2.3846	0.0790	1449.18	240
4		50	400	80	361798	1029.46	2.3846	0.0702	1424.56	232
5		50	500	99	359186	1029.46	2.3846	0.0657	1416.76	232
6	BBO	50	100	20	376528	1029.46	2.3846	0.1290	1522.88	256
7		100	100	45	376476	1029.46	2.3846	0.1311	1493.44	256
8		150	100	78	375143	1029.35	2.3846	0.1082	1497.23	240
9		200	100	130	371019	1029.46	2.3846	0.1061	1465.06	240
10		300	100	231	369848	1029.44	2.3846	0.0840	1485.37	240
11	BBO *linear*	50	100	20	383425	1029.46	2.3846	0.1624	1535.01	248
12		50	200	40	376956	1029.46	2.3846	0.1162	1484.97	248
13		50	300	59	370040	1029.46	2.3846	0.0846	1475.51	248
14		50	400	79	367116	1029.46	2.3846	0.0792	1458.29	248
15		50	500	99	366288	1029.46	2.3846	0.0787	1454.17	248
16	BBO p=0.01	50	100	22	376528	1029.46	2.3846	0.1290	1522.88	256
17	BBO p=0.02	50	100	21	382356	1029.46	2.3846	0.1385	1526.32	248
18	BBO p=0.05	50	100	25	395129	1029.46	2.3846	0.1310	1624.13	264
19	BBO p=0.10	50	100	26	398348	1029.46	2.3846	0.1367	1622.45	280
20	BBO p=0.15	50	100	30	400383	1029.46	2.3846	0.1597	1682.84	280

Table 5.7 shows another set of test cases that were created to study the behavior of the BBO algorithm under various parameters. The number of generations, the population size, the shape of the migration functions and the mutation probability of candidate solutions have been varied (in that order), so study their impact on solution quality. All of these experiments rely on the same problem instance in order to create comparable results.

According to these simulations, increasing the population size is inferior to increasing the number of generations (see test cases 1-5 vs. 6-10 in Table 5.7). In other words, generating only a small set of solutions and letting them compete against each other, yielded better results, than simply exploring more solutions within the same generation.

Linear migration functions seem to yield worse results than sinusoidal (test cases 1-5 vs. 11-15). Further, increasing the mutation probability (test cases 16-20), also yielded worse solutions. This indicates that the mutation procedure that alters existing solutions may actually destroy good solutions more frequently than actually yielding new and good results.

5.4.4 Comparison BBO and GA

Table 5.8: Results for specific problem instances where the number of VMs and users is lower than the number of PMs (test cases 1-8) and where the number of VMs and users is equal to or greater than the number of PMs (test cases 9-16).

No.	Alg.	nvms	npms	nusers	popsize	ngen	f_1	f_2	f_3	f_4	f_5	f_6
1	BBO	5	50	5	50	300	1701	112.2936	0.1879	0.0003	7.0499	2000
2	BBO	10	100	10	50	300	10768	228.4602	0.3354	0.0026	22.6339	1000
3	BBO	20	200	20	50	300	56085	458.0906	1.3929	0.0104	56.6255	500
4	BBO	30	300	30	50	300	138355	681.8566	2.3446	0.0161	94.7629	333
5	GA	5	50	5	50	300	1853	112.2936	0.1879	0.0003	7.5923	2000
6	GA	10	100	10	50	300	10974	228.4602	0.3354	0.0038	24.0459	1000
7	GA	20	200	20	50	300	57956	458.1627	1.3929	0.0105	58.4265	500
8	GA	30	300	30	50	300	140794	681.8566	2.3446	0.0255	95.9497	333
9	BBO	20	20	20	50	300	48120	457.9376	1.3929	0.0240	500.5000	600
10	BBO	40	20	40	50	300	213120	807.2144	1.4129	0.0615	2128.1434	512
11	BBO	80	20	80	50	300	973138	1595.5620	3.7507	0.2131	10050.3093	503
12	BBO	100	20	100	50	300	1596619	1947.6685	3.7480	0.4062	17449.3403	502
13	GA	20	20	20	50	300	48865	457.9376	1.3929	0.0228	501.4472	600
14	GA	40	20	40	50	300	210885	807.2744	1.4129	0.0614	2031.3212	525
15	GA	80	20	80	50	300	965028	1595.7259	3.7507	0.2094	9973.1556	500
16	GA	100	20	100	50	300	1602569	1947.4929	3.7480	0.4027	17070.7492	502

Table 5.8 shows results for a special subset of problem instances. The first half, test cases 1-8, shows results for instances where the number of VMs `nvms` and the number of users `nusers` is a fraction of the number of PMs. The second half, test cases 9-16, show results, for instances where `nvms` and `nusers` is equal to or greater than `npms`. In the first set of test cases, BBO can outperform GA as it is able to find solutions that have equal or smaller values for each of the objective functions. In the second set of test cases, this is no longer the case. One explanation for this performance edge in the first set of test cases, are the differences in the length of the solution encodings. BBO seems to perform better (relatively to GA), when solutions have a short encoding. This may be due to the way new generations are derived in both algorithms. In GA, solutions are ranked according to their objective function. Two solutions are chosen and recombined using uniform crossover, where high ranking (well performing) solutions are chosen more frequently than low ranking solutions. In BBO, solutions are also ranked. Based on that ranking, immigration rate λ and emigration rate μ are calculated. Good solutions (solutions with high μ) are chosen more frequently to spread their individual variables among the generation. Bad solutions (solutions with high λ) are chosen more frequently as the target of those individual variables. This more refined scheme of BBO, seems to have a more positive impact on shorter encoded solutions.

5.4.5 Optimization using the Internet Topology

The graph that defines how PMs are connected to each other described in Section 5.1 was created randomly. However, this might not necessarily reflect the Internet's architecture,

which is why we conducted another set of experiments that rely on a more precise representation. The Positive-Feedback Preference (PFP) model [93] was selected for automatically generating an architecture, that mirrors the Internet's architecture more closely (see also Section 2.2.4). Table 5.9 compares the random graph and the PFP model in terms of some key graph measures defined in Section 2.2.6.

Table 5.9: Graph measures for graphs used in the experiments

Graph Measure	Main Analysis	PFP Graph
Number of Nodes	50	50
Number of Edges	613	117
Mean Shortest Path Length	18.34	40.05
Graph Radius	23	53
Graph Diameter	30	84
Mean Closeness Centrality	0.01	0.01
Mean Betweenness Centrality	12.24	43.74
Mean Degree Centrality	24.52	4.68

Both graphs have the same number of edges, but the PFP graph is much sparser than the randomly generated graph (117 vs. 613 edges). In the PFP graph, the graph radius and diameter are much larger, which means that their nodes have a much higher eccentricity. Further, mean degree centrality is much lower, and the mean shortest path lengths are higher which indicates that the PFP graph is much more spread than the random graph.

Table 5.10: Results using the PFP graph as underlying topology[†]. Best and mean of the results of the final iteration of the algorithms are shown.

Algorithm	f_1	f_2	f_3	f_4	f_5	f_6
Greedy	811978	1029.5	2.4	0.1693	35165	336
Mean	*835893*	*1305.5*	*8.7*	*0.1793*	*39083*	*356*
BBO	529944	1029.5	2.4	0.0680	10646	232
Mean	*635796*	*1508.8*	*13.4*	*0.1624*	*19643*	*495*
GA	537487	1029.4	2.4	0.0533	11466	240
Mean	*644687*	*1479.3*	*12.7*	*0.1639*	*20223*	*491*

[†] Test parameters were `nvms`=50, `npms`=50, `nusers`=50 and for the two genetic algorithms additional parameters were `popsize`=50 and `ngen`=300.

Both BBO and GA outperform the Greedy algorithm in most of the objective functions, except for functions f_2 where Greedy and BBO are tied and f_3 where all of the algorithms are tied. The performance of BBO and GA is relatively equal, with only very small differences. The means shown in Table 5.10 are sometimes better (that is, lower) for the Greedy approach. This may be explained due to the fact that the genetic algorithms

introduce some random variability in each of their iterations. As a consequence, they are able to escape local minima in the search for new solutions, whereas the Greedy approach might be stuck in a local minimum indefinitely.

5.5 Limitations and Future Work

As mentioned in Section 1.1, we analyzed this assignment problem assuming that we have a guaranteed network bandwidth at our disposal. Related to that, in Section 4.2 we stated, that in order to calculate the delay between two PMs, we use the shortest path between them. Using these two assumptions means that we largely avoided the intricacies of network-related issues. In a real-world setup however, these assumptions may not hold and a more detailed network model that includes a more sophisticated routing scheme may be more suitable.

BBO is very versatile and does not rely on a particular network layout such as a star or tree topology. This is why our network topology in Section 5.1 is a randomly generated graph, where each node, on average, is adjacent to half of the other nodes and the graph generated such that it is guaranteed to be connected. In Section 5.4.5 we chose a different approach based on a model that is more accurately reflecting the Internet's topology. Future work could investigate whether the performance of BBO varies, if a topology is chosen that specifically represents the IoT's infrastructure. Other researchers, for example, scattered PMs on a hexagonal grid across a geographical area, which resulted in a PM having between 3 (the outer most PMs) and 6 neighbors (the inner ones) [52].

Another research direction could be to incorporate more heterogeneity into the system model to mimic real-world scenarios more closely. Then, one could explore the influences of these heterogeneities on the algorithm performance and solution quality. In this work, we included different machine availabilities into our model but limited our evaluation to identically constructed PMs. In future work, the impact of EDCs with different CPU and memory capabilities could be examined. Distributed data storage facilities could also come into play. Also, one could devise a system model where edge computing resources are not fixed but dynamically rented from some service provider: Different service providers could charge different hourly rates for the machines they provide and change these rates according to market fluctuations, similar to what we see today in the field of cloud computing providers.

5.6 Conclusion

The Internet of Things is growing at a rapid pace and poses new challenges that need to be tackled. An estimated 8.38 billion devices will be connected to the Internet by the end of 2017 and that number is expected to more than double by the end of 2020 [4]. We envisioned an architecture, where service providers create data centers attached to already existing base stations. These data centers are capable of accepting user requests and running user applications, while being physically closer to the users than remote

cloud data centers. Efficiently assigning both users and their applications to these data centers is complicated because latency requirements of users on one hand and cost considerations of service providers on the other, need to be balanced.

Biogeography-based Optimization is a metaheuristic that can be used to solve assignment problems. Given a set of initial solutions, it ranks the solutions depending on their quality, then alters and recombines them to arrive at new and better solutions. It does so over many iterations until a predefined number of solutions has been produced or the solutions' qualities are sufficient. BBO was adapted to handle the discrete nature of this problem as well as multiple objective functions.

In a first set of experiments, we minimized each of the objective functions individually. A maximum number of 300 generations was chosen. For four of the used objective functions, BBO converged towards an optimum. For the remaining two objective functions, the algorithm was unable to improve the initially chosen greedy solutions. This may be due to the imposed physical constraints of the problem and the limited information the algorithm was given to find solutions. In another set of experiments, the six objective functions were minimized simultaneously. Not all of the objective functions converged (similar to before), yet for some it was possible to find an improvement. Two more sets of experiments investigated the algorithm's performance using different starting parameters of BBO and a different underlying graph structure of the problem.

In all experiments, BBO generally outperformed the greedy approach as it found equally good or better solutions for the problem instances depending on the observed cost function. The performance advantage of BBO compared to GA is much smaller however (only a few percentage points). In the majority of scenarios, both algorithms deliver the same solution qualities, yet in some test cases BBO yielded slightly better solutions. For instances with many PM compared to users and VMs, BBO with its more refined solution recombination scheme, seems to have a clearer edge over GA.

The work shows that BBO is an admissible approach for solving this assignment problem encountered in Fog Computing which can lead to lower latency for user applications and cost savings and higher availability guarantees for service providers.

List of Figures

List of Tables

Bibliography

[1] Amazon found every 100ms of latency cost them 1% in sales. https://blog.gigaspaces.com/amazon-found-every-100ms-of-latency-cost-them-1-in-sales/. Accessed: 2017-03-31.

[2] Amazon Web Services. https://aws.amazon.com/. Accessed: 2017-03-27.

[3] Apple's Siri. https://www.apple.com/ios/siri/. Accessed: 2017-08-19.

[4] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://www.gartner.com/newsroom/id/3598917. Accessed: 2017-08-18.

[5] Google Cloud Platform. https://cloud.google.com/. Accessed: 2017-03-27.

[6] Gurobi Optimizer. http://www.gurobi.com/index. Accessed: 2017-09-12.

[7] How Apple's Siri really works. http://www.zdnet.com/article/how-apples-siri-really-works/. Accessed: 2017-08-19.

[8] Hybrid Invasive Weed / Biogeography-Based Optimization Scripts. http://embeddedlab.csuohio.edu/BBO/IWO.html. Accessed: 2017-08-08.

[9] IBM Bluemix. https://www.ibm.com/cloud-computing/bluemix/cloud-servers. Accessed: 2017-03-27.

[10] Internet of Things in Logistics. http://www.dhl.com/en/about_us/logistics_insights/dhl_trend_research/internet_of_things.html. Accessed: 2017-08-18.

[11] Matlab Documentation for Graph Centrality Measures. https://de.mathworks.com/help/matlab/ref/graph.centrality.html. Accessed: 2017-09-14.

[12] Microsoft Azure. https://azure.microsoft.com/. Accessed: 2017-03-27.

[13] OpenFlow(R) Switch Specification Ver 1.5.1. https://www.opennetworking.org/software-defined-standards/specifications/. Accessed: 2017-10-08.

[14] Original Biogeography-Based Optimization Scripts. http://academic.csuohio.edu/simond/bbo/. Accessed: 2017-08-08.

[15] Python 3.6 Programming Language. https://www.python.org/. Accessed: 2017-10-30.

[16] Senderkataster, Österreich. http://www.senderkataster.at/karte. Accessed: 2017-08-07.

[17] SPECpower benchmark. https://www.spec.org/power_ssj2008/. Accessed: 2017-08-06.

[18] These are six centrality measures on the same graph. https://commons.wikimedia.org/wiki/File:6_centrality_measures.png. Accessed: 2017-09-14.

[19] Ali Abbasi-Tadi, Mohammad Reza Khayyambashi, and Hadi Khosravi-Farsani. Data Center Task Scheduling Through Biogeography-Based Optimization Model With the Aim of Reducing Makespan. In *Computer and Knowledge Engineering (ICCKE), 2016 6th International Conference on*, pages 41–47. IEEE, 2016.

[20] Ravindra K Ahuja, James B Orlin, and Ashish Tiwari. A greedy genetic algorithm for the quadratic assignment problem. *Computers & Operations Research*, 27(10):917–934, 2000.

[21] Hafiz Munsub Ali and Daniel C Lee. A Biogeography-Based Optimization Algorithm for Energy Efficient Virtual Machine Placement. In *Swarm Intelligence (SIS), 2014 IEEE Symposium on*, pages 1–6. IEEE, 2014.

[22] Albert-László Barabási and Réka Albert. Emergence of Scaling in Random Networks. *science*, 286(5439):509–512, 1999.

[23] James C Bean. Genetic Algorithms and Random Keys for Sequencing and Optimization. *ORSA journal on computing*, 6(2):154–160, 1994.

[24] Anton Beloglazov, Jemal Abawajy, and Rajkumar Buyya. Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing. *Future generation computer systems*, 28(5):755–768, 2012.

[25] Anton Beloglazov and Rajkumar Buyya. Optimal Online Deterministic Algorithms and Adaptive Heuristics for Energy and Performance Efficient Dynamic Consolidation of Virtual Machines in Cloud Data Centers. *Concurrency and Computation: Practice and Experience*, 24(13):1397–1420, 2012.

[26] Theophilus Benson, Aditya Akella, and David A Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[27] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation-Volume 2*, pages 273–286. USENIX Association, 2005.

[28] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[29] P Di Barba, F Dughiero, ME Mognaschi, A Savini, and S Wiak. Biogeography-inspired Multiobjective Optimization and MEMS Design. *IEEE Transactions on Magnetics*, 52(3):1–4, 2016.

[30] Nick G Duffield, Pawan Goyal, Albert Greenberg, Partho Mishra, Kadangode K Ramakrishnan, and Jacobus E van der Merive. A Flexible Model for Resource Management in Virtual Private Networks. In *ACM SIGCOMM Computer Communication Review*, volume 29, pages 95–108. ACM, 1999.

[31] Waleed Ejaz, Muhammad Naeem, Adnan Shahid, Alagan Anpalagan, and Minho Jo. Efficient energy management for the internet of things in smart cities. *IEEE Communications Magazine*, 55(1):84–91, 2017.

[32] Md Hasanul Ferdaus, Manzur Murshed, Rodrigo N Calheiros, and Rajkumar Buyya. Virtual Machine Consolidation in Cloud Data Centers using ACO Metaheuristic. In *European Conference on Parallel Processing*, pages 306–317. Springer, 2014.

[33] Yongqiang Gao, Haibing Guan, Zhengwei Qi, Yang Hou, and Liang Liu. A Multi-Objective Ant Colony System Algorithm for Virtual Machine Placement in Cloud Computing. *Journal of Computer and System Sciences*, 79(8):1230–1242, 2013.

[34] Michael R Garey, Ronald L Graham, David S Johnson, and Andrew Chi-Chih Yao. Resource Constrained Scheduling as Generalized Bin Packing. *Journal of Combinatorial Theory, Series A*, 21(3):257–298, 1976.

[35] José Fernando Gonçalves and Mauricio GC Resende. Biased Random-Key Genetic Algorithms for Combinatorial Optimization. *Journal of Heuristics*, 17(5):487–525, 2011.

[36] Lin Gu, Deze Zeng, Song Guo, Ahmed Barnawi, and Yong Xiang. Cost Efficient Resource Management in Fog Computing Supported Medical Cyber-Physical System. *IEEE Transactions on Emerging Topics in Computing*, 5(1):108–119, 2017.

[37] Jochen W Guck, Amaury Van Bemten, and Wolfgang Kellerer. DetServ: Network Models for Real-Time QoS Provisioning in SDN-based Industrial Environments. *IEEE Transactions on Network and Service Management*, 2017.

[38] Chuanxiong Guo, Guohan Lu, Helen J Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. Secondnet: a data center network virtualization architecture with bandwidth guarantees. In *Proceedings of the 6th International COnference*, page 15. ACM, 2010.

[39] Harshit Gupta, Amir Vahid Dastjerdi, Soumya K Ghosh, and Rajkumar Buyya. iFogSim: A Toolkit for Modeling and Simulation of Resource Management Techniques in the Internet of Things, Edge and Fog Computing Environments. *Software: Practice and Experience*, 47(9):1275–1296, 2017.

[40] Hamed Haddadi, Miguel Rio, Gianluca Iannaccone, Andrew Moore, and Richard Mortier. Network topologies: inference, modeling, and generation. *IEEE Communications Surveys & Tutorials*, 10(2), 2008.

[41] P Ji, Yongzhong Wu, and Haozhao Liu. A Solution Method for the Quadratic Assignment Problem (QAP). In *The Sixth International Symposium on Operations Research and Its Applications (ISORA'06), Xinjiang, China, August*, pages 8–12, 2006.

[42] Mike Jia, Jiannong Cao, and Weifa Liang. Optimal Cloudlet Placement and User to Cloudlet Allocation in Wireless Metropolitan Area Networks. *IEEE Transactions on Cloud Computing*, 2015.

[43] Hao Jin, Deng Pan, Jing Xu, and Niki Pissinou. Efficient VM Placement with Multiple Deterministic and Stochastic Resources in Data Centers. In *Global Communications Conference (GLOBECOM), 2012 IEEE*, pages 2505–2510. IEEE, 2012.

[44] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Data Center Traffic: Measurements & Analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 202–208. ACM, 2009.

[45] Ahmad A Kardan, Atena Kavian, and Amir Esmaeili. Simultaneous feature selection and feature weighting with k selection for knn classification using bbo algorithm. In *Information and Knowledge Technology (IKT), 2013 5th Conference on*, pages 349–354. IEEE, 2013.

[46] Gholamreza Khademi, Hanieh Mohammadi, and Dan Simon. Hybrid invasive weed/biogeography-based optimization. *Engineering Applications of Artificial Intelligence*, 64:213–231, 2017.

[47] Scott Kirkpatrick, C Daniel Gelatt, Mario P Vecchi, et al. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[48] Tjalling C Koopmans and Martin Beckmann. Assignment Problems and the Location of Economic Activities. *Econometrica: journal of the Econometric Society*, pages 53–76, 1957.

[49] Kyriakos Kritikos, Barbara Pernici, Pierluigi Plebani, Cinzia Cappiello, Marco Comuzzi, Salima Benrernou, Ivona Brandic, Attila Kertész, Michael Parkin, and Manuel Carro. A Survey on Service Quality Description. *ACM Computing Surveys (CSUR)*, 46(1):1, 2013.

[50] Karthik Kumar, Jibang Liu, Yung-Hsiang Lu, and Bharat Bhargava. A survey of computation offloading for mobile systems. *Mobile Networks and Applications*, 18(1):129–140, 2013.

[51] Etienne Le Sueur and Gernot Heiser. Dynamic voltage and frequency scaling: The laws of diminishing returns. In *Proceedings of the 2010 international conference on Power aware computing and systems*, pages 1–8, 2010.

[52] Chanh Nguyen Le Tan, Cristian Klein, and Erik Elmroth. Location-aware load prediction in edge data centers. In *Fog and Mobile Edge Computing (FMEC), 2017 Second International Conference on*, pages 25–31. IEEE, 2017.

[53] Jeongkeun Lee, Yoshio Turner, Myungjin Lee, Lucian Popa, Sujata Banerjee, Joon-Myung Kang, and Puneet Sharma. Application-driven bandwidth guarantees in datacenters. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 467–478. ACM, 2014.

[54] Wee Loon Lim, Muhammad A'rif Shah Alias, and Habibollah Haron. A hybrid metaheuristic for the generalized quadratic assignment problem. In *Research and Development (SCOReD), 2015 IEEE Student Conference on*, pages 467–471. IEEE, 2015.

[55] Fabio Lopez-Pires and Benjamín Barán. Virtual machine placement literature review. *arXiv preprint arXiv:1506.01509*, 2015.

[56] Dražen Lučanin and Ivona Brandic. Pervasive Cloud Controller for Geotemporal Inputs. *IEEE Transactions on Cloud Computing*, 4(2):180–195, 2016.

[57] Haiping Ma. An analysis of the equilibrium of migration models for biogeography-based optimization. *Information Sciences*, 180(18):3444–3464, 2010.

[58] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *SIGCOMM Comput. Commun. Rev.*, pages 69–74, 2008.

[59] Peter Mell, Tim Grance, et al. The NIST Definition of Cloud Computing. 2011.

[60] Ruben Mennes, Bart Spinnewyn, Steven Latré, and Juan Felipe Botero. GRECO: A Distributed Genetic Algorithm for Reliable Application Placement in Hybrid Clouds. In *Cloud Networking (Cloudnet), 2016 5th IEEE International Conference on*, pages 14–20. IEEE, 2016.

[61] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT press, 1998.

[62] Nitinder Mohan and Jussi Kangasharju. Edge-Fog Cloud: A Distributed Cloud For Internet of Things Computations. In *Cloudification of the Internet of Things (CIoT)*, pages 1–6. IEEE, 2016.

[63] Yurii Nesterov. *Introductory lectures on convex optimization: A basic course*, volume 87. Springer Science & Business Media, 2013.

[64] Ben Pfaff, Justin Pettit, Keith Amidon, Martin Casado, Teemu Koponen, and Scott Shenker. Extending Networking into the Virtualization Layer. In *Hotnets*, 2009.

[65] Fabio López Pires and Benjamín Barán. A virtual machine placement taxonomy. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 159–168. IEEE, 2015.

[66] Lucian Popa, Praveen Yalagandula, Sujata Banerjee, Jeffrey C Mogul, Yoshio Turner, and Jose Renato Santos. ElasticSwitch: Practical Work-Conserving Bandwidth Guarantees for Cloud Computing. *ACM SIGCOMM Computer Communication Review*, 43(4):351–362, 2013.

[67] MATLAB R2016b. The MathWorks Inc., Natick, Massachusetts, 2016.

[68] Yi Ren, Junichi Suzuki, Athanasios Vasilakos, Shingo Omura, and Katsuya Oba. Cielo: An Evolutionary Game Theoretic Framework for Virtual Machine Placement in Clouds. In *Future Internet of Things and Cloud (FiCloud), 2014 International Conference on*, pages 1–8. IEEE, 2014.

[69] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A Taxonomy and Survey of Cloud Computing Systems. *INC, IMS and IDC*, pages 44–51, 2009.

[70] Xiaojun Ruan and Haiquan Chen. Performance-to-Power Ratio Aware Virtual Machine (VM) Allocation in Energy-Efficient Clouds. In *Cluster Computing (CLUSTER), 2015 IEEE International Conference on*, pages 264–273. IEEE, 2015.

[71] Maurice Rüegg. Virtual Private Network Provisioning in the Hose Model. 2003.

[72] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[73] Dan Simon. Biogeography-based optimization. *IEEE transactions on evolutionary computation*, 12(6):702–713, 2008.

[74] Edward H Simpson. Measurement of diversity. *Nature*, 1949.

[75] James E Smith and Ravi Nair. The Architecture of Virtual Machines. *Computer*, 38(5):32–38, 2005.

[76] Ying Song, Min Liu, and Zheng Wang. Biogeography-Based Optimization for the Traveling Salesman Problems. In *Computational Science and Optimization (CSO), 2010 Third International Joint Conference on*, volume 1, pages 295–299. IEEE, 2010.

[77] Benjamin Speitkamp and Martin Bichler. A Mathematical Programming Approach for Server Consolidation Problems in Virtualized Data Centers. *IEEE Transactions on services computing*, 3(4):266–278, 2010.

[78] Bart Spinnewyn, Bart Braem, and Steven Latré. Fault-tolerant Application Placement in Heterogeneous Cloud Environments. In *Network and Service Management (CNSM), 2015 11th International Conference on*, pages 192–200. IEEE, 2015.

[79] Jianyun Sun, Haopeng Chen, and Zhida Yin. AERS: An Autonomic and Elastic Resource Scheduling Framework for Cloud Applications. In *Services Computing (SCC), 2016 IEEE International Conference on*, pages 66–73. IEEE, 2016.

[80] David M Tate and Alice E Smith. A genetic approach to the quadratic assignment problem. *Computers & Operations Research*, 22(1):73–83, 1995.

[81] Karima Velasquez, David Perez Abreu, Marilia Curado, and Edmundo Monteiro. Service Placement for Latency Reduction in the Internet of Things. *Annals of Telecommunications*, 72(1-2):105–115, 2017.

[82] William Voorsluys, James Broberg, and Rajkumar Buyya. Introduction to Cloud Computing. *Cloud computing: Principles and paradigms*, pages 1–41, 2011.

[83] Douglas B. West. *Introduction to Graph Theory*. Prentice Hall, 2000.

[84] Grant Wu, Maolin Tang, Yu-Chu Tian, and Wei Li. Energy-Efficient Virtual Machine Placement in Data Centers by Genetic Algorithm. In *Neural Information Processing*, pages 315–323. Springer, 2012.

[85] Yongqiang Wu, Maolin Tang, and Warren Fraser. A simulated annealing algorithm for energy efficient virtual machine placement. In *Systems, Man, and Cybernetics (SMC), 2012 IEEE International Conference on*, pages 1245–1250. IEEE, 2012.

[86] Jing Xu and Jose AB Fortes. Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments. In *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pages 179–188. IEEE Computer Society, 2010.

[87] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog computing: Platform and applications. In *Hot Topics in Web Systems and Technologies (HotWeb), 2015 Third IEEE Workshop on*, pages 73–78. IEEE, 2015.

[88] Minyi Yue. A simple proof of the inequality FFD(L) 11/9 OPT(L) + 1 for the FFD bin-packing algorithm. *Acta Mathematicae Applicatae Sinica (English Series)*, 7(4):321–331, 1991.

[89] Yudong Zhang, Preetha Phillips, Shuihua Wang, Genlin Ji, Jiquan Yang, and Jianguo Wu. Fruit Classification by Biogeography-Based Optimization and Feedforward Neural Network. *Expert Systems*, 33(3):239–253, 2016.

[90] Qinghua Zheng, Rui Li, Xiuqi Li, Nazaraf Shah, Jianke Zhang, Feng Tian, Kuo-Ming Chao, and Jia Li. Virtual Machine Consolidated Placement Based On Multi-Objective Biogeography-Based Optimization. *Future Generation Computer Systems*, 54:95–122, 2016.

[91] Qinghua Zheng, Rui Li, Xiuqi Li, and Jie Wu. A Multi-Objective Biogeography-Based Optimization for Virtual Machine Placement. In *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pages 687–696. IEEE, 2015.

[92] Shi Zhou and Raúl J Mondragón. The Rich-Club Phenomenon in the Internet Topology. *IEEE Communications Letters*, 8(3):180–182, 2004.

[93] Shi Zhou and Raúl J Mondragón. The Positive-Feedback Preference Model of the AS-level Internet Topology. In *Communications, 2005. ICC 2005. 2005 IEEE International Conference on*, volume 1, pages 163–167. IEEE, 2005.