D-CACEV: a Dynamic Cost And Carbon Emission-efficient Virtual machine placement method for green distributed clouds

Ehsan Ahvar, Shohreh Ahvar, Zoltan Adam Mann, Noel Crespi, Joaquin Garcia-Alfaro, Roch Glitho

Authors (draft) version
This paper has been accepted in IEEE Access
D-CACEV: a Dynamic Cost and Carbon Emission-Efficient Virtual Machine Placement Method for Green Distributed Clouds

Ehsan Ahvar, Shohreh Ahvar, Zoltan Adam Mann,
Noel Crespi, Joaquin Garcia-Alfaro, and Roch Glitho,

Abstract—As an increasing amount of data processing is done at the network edge, high energy costs and carbon emission of Edge Clouds (ECs) are becoming significant challenges. The placement of application components (e.g., in the form of containerized microservices) on ECs has an important effect on the energy consumption of ECs, impacting both energy costs and carbon emissions. Due to the geographic distribution of ECs, there is a variety of resources, energy prices and carbon emission rates to consider, which makes optimizing the placement of applications for cost and carbon efficiency even more challenging than in centralized clouds. This paper presents a Dynamic Energy cost and Carbon emission-efficient Application placement method (DECA) for green ECs. DECA addresses both the initial placement of applications on ECs and the re-optimization of the placement using migrations. DECA considers geographically varying energy prices and carbon emission rates as well as optimizing the usage of both network and computing resources at the same time. By combining a prediction-based A* algorithm with Fuzzy Sets technique, DECA makes intelligent decisions to optimize energy cost and carbon emissions. Simulation results show the applicability and performance of DECA.

Index Terms—Edge cloud, Energy consumption, Energy costs, Green computing, Carbon emission, Application placement.

1 Introduction

The Internet of Things (IoT) is producing rapidly increasing amounts of data. Data analytics applications that process IoT data require significant computational capacity, which IoT devices typically do not possess. Using centralized cloud data centers to host the analytics applications is an option, but transferring the data from the IoT devices to the cloud incurs high latency and large network traffic. Therefore, new distributed computing paradigms that move processing closer to the network edge (fog computing, edge computing etc.) are gaining popularity. In the computing model considered in this paper, computational resources are provided in several *Edge clouds (ECs)* instead of a single centralized cloud. ECs have limited capacity and are geographically distributed. Applications can be deployed on ECs, and data produced by IoT devices can be processed in an EC near the IoT devices. Thereby, latency and network traffic are significantly reduced, making ECs an attractive paradigm for many IoT applications [1], [2].

E. Ahvar is with Learning, Data and Robotics Lab, ESIEA, Paris, France. e-mail: (ehsan.ahvar@esiea.fr).
S. Ahvar is with ISEP-Institut Supérieur d'Électronique de Paris. e-mail: (shohreh.ahvar@isep.fr).
N. Crespi, J. Garcia-alfaro are with Télécom SudParis, Institut Polytechnique de Paris, France. e-mail: (noel.crespi, joaquin.garcia_alfaro@telecom-sudparis.eu).
Z. A. Mann is with University of Duisburg-Essen, Germany. e-mail: (zoltan.mann@paluno.uni-due.de).
R. Glitho is with Concordia University, Canada. e-mail: (glitho@ciise.concordia.ca).

With the rise of data processing in ECs, the increasing energy consumption of ECs is becoming a major concern for two reasons: energy costs and carbon emissions. Both energy costs and carbon emissions are becoming pressing issues for the providers of ECs [10].

Similar to centralized clouds, also ECs need energy saving methods, e.g., workload consolidation. However, minimizing energy costs and carbon emissions in ECs is a more complex problem. Energy prices and carbon emission rates vary by location and even by time (e.g., because of different local energy sources). Therefore, even the same energy consumption may lead to different energy costs and carbon emissions depending on which EC (and when) serves the given workload. It is important to note that there is no correlation between the cleanness (carbon footprint) and the price of a location's energy sources [3]. Hence, optimizing energy costs and carbon emissions are two independent objectives.

We consider an infrastructure comprising several geographically distributed ECs, where each EC consists of a set of Compute Nodes (CNs). A set of applications is to be placed on this infrastructure. Every application consists of one or more components, for example in the form of containerized microservices. For every component, detecting an appropriate place (in which of the ECs, on which CN) is considered as an important issue. For the component placement, there is a variety of resources, energy prices and carbon emission rates to consider. To optimize energy costs and carbon emissions, we have three levers: (i) minimizing energy consumption (usually by optimizing resource utilization), (ii) choosing resources in locations with low energy price, and

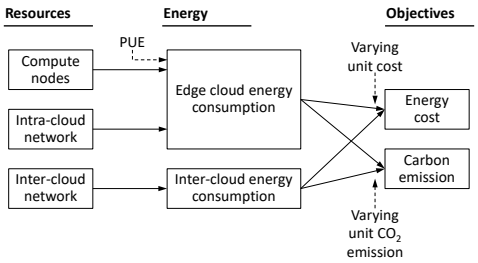


Fig. 1. Sources of energy consumption and its impact on optimization objectives

(iii) choosing resources in locations with low carbon emission rate. Hence the challenge is to consider these three, sometimes conflicting aims simultaneously (see also Fig. 1).

To address the above problem, this paper proposes a dynamic energy cost and carbon emission-efficient application placement method (DECA) for ECs, considering the above three levers to optimize both energy costs and carbon emissions in distributed ECs. DECA includes two main parts: (i) determining the initial placement of newly deployed applications and (ii) re-optimization of the placement of applications to react to workload changes. In contrast to most previous works, DECA considers both CNs and network devices because both of them may contribute significantly to energy consumption.

DECA combines a variant of the A* search algorithm [6] with a Fuzzy Sets technique [7]. Using these powerful techniques, DECA can perform more effective optimization than traditional greedy heuristics used by most existing approaches [8], [9]. We describe in the Appendix the benefit of the A* algorithm for application placement in ECs compared to other heuristics.

DECA performs joint optimization of compute and network resources, also considering their associated energy price and carbon emission rate. It can select CNs from multiple ECs to place the components of an application, in order to (i) be able to achieve low overall energy cost and carbon emission and (ii) overcome capacity limitations of a single EC.

Our major contributions are summarized below:

1) We build this work based on our previous work on green cloud computing [10], adapting it to the characteristics of emerging EC systems and improving its application placement method (i.e., from static placement to dynamic).
2) To offer a better view about DECA mechanism, a comprehensive logical architecture is presented.
3) New methods for the dynamic re-optimization of the placement using live migration are proposed. Two AC migration mechanisms are proposed for under-utilized and over-utilized CNs respectively.
4) We perform a comprehensive comparison on energy cost, carbon footprint and energy consumption for different AC placement algorithms.

The rest of the paper is organized as follows. Section 2 describes related work. Section 3 introduces our problem formulation. Section 4 provides the algorithmic solution underlying DECA. Section 5 evaluates DECA. Section 6 concludes the paper.

2 Related work

As already mentioned, the applications are assumed to be made of independently deployable ACs, for example in the form of VM-based or containerized microservices. The provider should make a decision on resource allocation for the components by selecting the most suitable CN. This process is known as placement.

In recent years, many different aspects of the placement problem (mostly in the form of VM placement) have been investigated [9]. We can divide the related work into three categories: i) the works which focus on improving energy consumption and cost, ii) the studies which consider both energy consumption/cost and carbon emission together, iii) the works that utilize AC or VM migration algorithms to improve energy consumption and/or carbon emission.

2.1 Energy consumption and cost

Li et al. [11] considered network and compute resources at the same time for their allocation algorithm. Pahlevan et al. [12] proposed an energy- and network-aware approach that integrates heuristic and machine learning methods. You et al. [13] designed a network-aware VM placement method to improve communication cost. Although these works considered also the optimization of data transfer, they are limited to a single-node (i.e., centralized cloud) environment and are not appropriate for a distributed EC with varying resource prices and carbon emission rates.

Goudarzi et al. [47] have recently proposed an application placement technique based on the Memetic Algorithm to make batch application placement decision for IoT applications in a heterogeneous Edge and Fog computing environment. However, in this work, the energy consumption is considered from the IoT device perspective. They also do not consider the price of energy.

Pallewatta et al. [48] proposed a microservices-based IoT application placement technique for heterogeneous and resource constrained fog environments. They also proposed a fog node architecture to support their proposed placement approach. But their main objective is to minimize latency and network usage, and not energy costs.

Hu et al. [49] proposed an approach to optimize the placement of service-based applications in clouds for reducing the inter-machine traffic. They first partition the application into several parts while trying to keep the overall traffic between the created parts to a minimum. Then, the created parts are carefully located into machines with respect to their resource and traffic demands. However, they do not consider energy cost optimisation.

Hassan et al. [50] have recently proposed a method for service placement in fog-cloud systems. They classified services into two categories: critical and normal ones. For critical services, they try to minimize response time, and for normal ones the goal is to reduce the energy consumption of the fog environment. The same authors [52] formulated the VM placement of IoT applications in Cloud DCs as an MILP model and proposed two algorithms. Minimization of the power (i.e., CPU) and network for the first algorithm. And second algorithm focuses on reducing the power (i.e., CPU) and resources wastage. Yet, carbon emission and energy cost optimisation has not been accounted for in these two works.

Kayal et al. [51] proposed a placement strategy to jointly optimize energy consumption of fog nodes and communication costs of applications. The algorithm applies the Markov approximation

method to solve the combinatorial optimization objective. However, they are concerned with methods for assigning microservices to fog nodes with the objective of balancing energy consumption at fog nodes and network traffic costs.

Nabavi et al. [53] proposed a multi-objective VM placement scheme (considering VMs as fog tasks) for edge cloud DCs called TRACTOR using an artificial bee colony optimization algorithm. TRACTOR goal is power and network-aware assignment of VMs onto PMs. The proposed scheme aims to minimize the network traffic of the interacting VMs and the power dissipation of the DC's switches and PMs.

Our current work is different from all above-mentioned results since we consider not only energy costs but also carbon emission.

2.2 Both energy consumption (cost) and carbon emission

Khosravi et al. [16] considered both carbon emission and energy during VM allocation. However, they did not consider the variability of energy prices. Also, inter-VM (inter-AC) communication was not considered, although it is an important factor for reducing energy of network resources. Khosravi et al. in [17] have presented several energy and carbon-aware algorithms. But they did not consider the energy consumed by network elements in their energy model. Zhou et al. [4] jointly considered electricity cost, emission and Service Level Agreement (SLA) reduction for distributed ECs, while Gu et al. [18] presented a method to minimize carbon emission of ECs or Data Centers (DCs) while satisfying constraints on response time, electricity budget and maximum number of running CNs in an environment with homogeneous CNs. These papers target service jobs with constraints on response time. Moreover, different from all mentioned works, our work considers both component (VM) placement and migration to keep costs and emissions low.

2.3 AC (VM) migration

Tziritas et al. [19] targeted the problem of placement considering two objectives: (1) to minimize energy consumption of the CNs, and (2) to minimize the network overhead stemming from communication between VMs and from VM migrations. They select the most energy-consuming CN based on both maximum power and current workload to migrate its VMs. This method may select CNs with low power consumption but heavy workload as migration source and CNs with high power consumption and low workload as destination, which may lead to sub-optimal results.

Zhou et al. [20] proposed a VM deployment algorithm called three-threshold energy saving algorithm (TESA) and five VM selection algorithms: MIMT, MAMT, HPGT, LPGT and RCT. Unlike our work which selects a destination CN based on both communication and compute resource metrics, they select a CN with the least increase of power consumption (we called it Min. Compute policy) due to VM allocation. This way, the target CN may be located far from the source CN so that migration consumes a large amount of energy.

Mustafa et al. [30] proposed two consolidation based techniques to reduce energy consumption along with resultant SLA violations. They also enhanced two existing techniques that attempt to reduce energy consumption and SLA violations. They finally show that the proposed techniques perform better than the selected heuristic based techniques in terms of energy, SLA, and migrations.

Zheng et al. [21] proposed a dynamic energy efficient resource allocation scheme. They consider a mapping probability matrix where each VM request is assigned with a probability on a specific CN. The proposed method then decides where to allocate new VM requests and whether to migrate existing VMs in order to improve energy efficiency. Although the authors proposed an idea to migrate VMs for energy reduction, their exact solution in situation of CN overloading is not clear.

Beloglazov et al. [22] devised heuristics to continuously consolidate VMs leveraging live migration and switching off idle CNs to minimize the number of utilized CNs. They proposed a modified version of the Best Fit Decreasing algorithm (MBFD) to place VMs on CNs and four heuristics to select VMs for migration. They did not consider the energy consumption of network elements. In another study, Beloglazov et al. [23] proposed three stages of VM placement optimization including reallocation based on current utilization of multiple system resources, optimization of virtual network topologies established between VMs and VM reallocation considering thermal state of the resources. However, Beloglazov et al. in both studies did not consider carbon emission.

Liu et al. [24] developed an ant colony system-based approach to reduce cloud energy consumption. To handle both homogeneous and heterogeneous CN environments, they also proposed an order exchange and migration mechanism. However, their objective is limited to minimizing the number of active CNs. Forestiero et al. [25] proposed a hierarchical method (i.e, composed of two workload assignment and migration algorithms) for efficient workload management in distributed ECs. Li et al. [26] proposed a dynamic virtual machine scheduling algorithm, called GRANITE, to minimize total EC energy consumption. However, these works do not take into account carbon emission.

Ibrahim et al. [54] attempted to reduce consumed energy, number of VM migrations, number of host shutdowns and the combined metric Energy SLA Violation (ESV) with a dynamic consolidation of VMs. In the method, the servers are checked periodically and appropriate VMs from under and over utilised servers are migrated to destinations selected based on Particle Swarm Optimization techniques. The authors mentioned that increasing energy consumption has a significant impact on the environment due to emissions of carbon. However, they have not done any study in carbon emission reduction. Cost also was not considered by the authors.

Table 1 lists the related works considering their main objectives/characteristics. All mentioned related works addressed some of the points listed in the Introduction to characterize the problem, but only in isolation, missing some other important aspects. Our previous work CACEV [10] is the first to address most aspects in combination, although in a cloud computing setting. It is also the first VM placement algorithm integrating the prediction-based A* search algorithm [6] with a Fuzzy Sets technique [7]. However, CACEV makes only a static VM placement. Our current work, DECA, extends CACEV to ECs and with support for VM migrations. Different from related works on VM consolidation, DECA uses a fuzzy set-based decision maker which can sharply improve its performance.

3 Problem Formulation

3.1 System model

To offer a realistic solution, the paper considers a system model characterized by the following points:

TABLE 1
Related Work Summary

Reference	energy saving/cost	carbon emission	AC/VM migration
[11], [12], [13], [47], [48], [49], [50], [51], [52], [53]	✓		
[4], [16], [17], [18]	✓	✓	
[19], [20], [21], [22], [23], [24], [25], [26], [54]	✓		✓

1) Heterogeneity of ECs, CNs, and network devices in terms of capacity and energy consumption characteristics.
2) Heterogeneity of application components in terms of resource needs.
3) Load-dependent energy consumption (for example, the energy consumption of a CN depends on its CPU load).
4) Joint optimization of compute and network resources.
5) Arbitrary network topology among ECs and within ECs; in particular, there can be multiple network paths between a pair of CNs.
6) Variety in unit energy price and unit carbon footprint among ECs.
7) Ability to select CNs from multiple ECs to place the components of an application.
8) Taking into account the communication between application components and the associated impact on network traffic, preferring to place components with intensive communication close to each other.
9) Workload consolidation by live migration of application components.

We consider a hierarchical distributed architecture [27] consisting of a set of ECs, with each EC consisting of a set of CNs. The ECs and the inter-cloud connectivity information are given by a graph $G_0 = (D, E, wD, wE)$ where D is the set of ECs, wD denotes their current capacity, E consists of connections (network paths) among the ECs, and wE denotes the weights of the connections (e.g., number of routers on the network paths). Each EC is characterized by a Power Usage Effectiveness (PUE) value and is associated with one or more energy sources with different energy prices and carbon footprint rates. PUE is considered as the ratio of total power consumed by the EC to the power consumed by IT devices within the EC [16]. We assume a high-capacity backbone network to carry the traffic between the ECs. Inside an EC, the model (and our proposed algorithm) supports both structured (e.g., Fat-Tree [38]) and arbitrary [28] topologies. Table 2 gives an overview of the abbreviations and notations used in the paper.

Each EC $d \in D$ is represented by a weighted graph $G_d = (N_d, E_d, wN_d, wE_d)$, where N_d is the set of CNs in EC d, E_d is the set of links (network paths) between CNs, wN_d shows the current capacity of the CNs, and wE_d denotes the link weights (e.g., number of switches on the network path) between CNs within the given EC. Similar to [29], for every pair of CNs i and i' in EC d, a set of pre-calculated paths from i to i' is considered, and is given by E_d. Resource parameters of each CN i are given as a vector R_i, including CPU, memory, disk, and I/O bandwidth.

To handle time-varying request rates and energy prices, time is split into equal time windows. We assume that within a time window T, energy prices do not change.

A set A of application deployment requests is received for the next time window. Application $a \in A$ consists of a set m_a of application components. The set of all requested application components is denoted by $M = \cup_{a \in A} m_a$. An application usually consists of several components that communicate to each other. An $|M| \times |M|$ traffic matrix TR contains the amount of traffic exchanged among the application components. Each application component k is characterized by a vector V_k of its resource needs according to CPU, memory, disk, and I/O bandwidth.

Carbon emission and energy cost are related to the amount of energy consumption by network and server resources. The energy consumption of a CN is considered as a function of its CPU load since the CPU is the main contributor to dynamic power consumption in a CN [31], [36]. Switches and routers are the main contributors to network energy consumption [37]. We consider sleep and active modes for both CNs [31] and switches [32], [33].

3.2 Application allocation problem

Each component in M has to be assigned to an EC, taking into account the geographically varying energy prices and carbon emission rates (e.g., see [46]). The distributed requests in each selected EC are then allocated on appropriate CNs in the EC. Appropriate paths are also selected between the CNs hosting communicating components.

As (1) shows, our aim is to allocate the application components such that energy costs and carbon emissions are minimized:

$$\begin{cases} \text{minimize: } (Y_{tot}, Z_{tot}), \text{ where} \\ Y_{tot} = Y_{cl} + Y_{com}, \text{ and} \\ Z_{tot} = Z_{cl} + Z_{com} \end{cases} \tag{1}$$

with the constraint that the selected CNs must have enough capacity to accommodate the components shown in (2).

$$\sum_{d \in D} \sum_{i \in N_d} (R_i \cdot S_d) \geq \sum_{k \in M} V_k. \tag{2}$$

In (1), Y_{tot} is the total cost, Z_{tot} is the total carbon emission, Y_{cl} is the cost within the ECs, Y_{com} is the cost of the inter-cloud network communication, Z_{cl} is the carbon emission within the ECs, and Z_{com} is the carbon emission of the inter-cloud network communication. In (2), V_k is a vector of the requested resources of component k, the variable S_d is 1 if EC d is selected for hosting some of the requested components (otherwise 0), and R_i is the capacity vector of CN i.

The next subsections describe the details of determining Y_{tot} and Z_{tot} (see also Fig. 2 for an overview).

3.2.1 Overall cost formulation (Y_{tot})

In order to determine Y_{tot}, (3)-(16) formulate Y_{cl} and (17)-(19) formulate Y_{com}.

Costs incurred within the ECs: Y_{cl} is the cost of incremental energy of selected ECs (including both the CNs and the intra-cloud network) to place the newly requested components:

$$Y_{cl} = \sum_{d \in D} PUE_d \cdot (Y'_d + Y''_d) \cdot y_d \cdot S_d. \tag{3}$$

TABLE 2
Notation overview

Symbol	Meaning
A	Set of applications to be deployed
AC	Application Component
CN	Compute Node
C_k	CPU load of component k
$C^i_{\max}$	CPU processing capacity of CN i
D	Set of Edge Clouds (ECs)
EC	Edge Cloud
E	Set of connections between ECs
E_d	Set of links among CNs in EC d
$E^i_{\text{inc},k}$	Energy increment of component k on CN i
E^i_{wp}	Energy for going from sleep to active mode
E^i_{idle}	Idle energy consumption of node i
E^i_k	Additional energy of running component k on node i
$E^i_{\max}$	Energy consumption of node i with full load
E^B_{inc}	Incremental energy of network element B for a packet
E^B_p	Per-packet processing energy of network element B
$E^B_{\text{S\&F}}$	Per-byte store&forward energy of network element B
m_a	Set of components of application a
M	Set of all components to be deployed
N_d	Set of compute nodes (CNs) in EC d
PUE_d	Power usage effectiveness of EC d
R_i	Resource vector of CN i
$SerEn_i$	Energy increment of new components on node i
TR	Traffic matrix of inter-component traffic
V_k	Resource needs of component k
VM	Virtual Machine
$wD(d)$	Current capacity of EC d
$wE(e)$	Weight of connection e
$W^t_{i,i'}$	Incremental energy for transferring the t^{th} packet
$wE_d(e)$	Weight of link e in EC d
$wN_d(i)$	Current capacity of CN i in EC d
Y_{tot}	Total cost
Y_{cl}	Cost incurred within the ECs
Y_{com}	Cost incurred by inter-cloud communication
y_d	Energy unit price for EC d
Y'_d	Energy increment of CNs in EC d
Y''_d	Energy increment of switches in EC d
$y_{d,d'}$	Energy unit price between ECs d and d'
$Y_{d,d'}$	Incremental energy between ECs d and d'
Z_{com}	Carbon emission of inter-cloud communication
Z_{tot}	Total carbon emission
Z_{cl}	Carbon emission generated within the ECs
$\lambda_{i,i',t}$	Path between nodes i and i' to carry the t^{th} packet
$\delta_{i,i'}$	Number of exchanged packets between nodes i and i'

Here, Y'_d and Y''_d are the incremental energy consumption of CNs and switches, respectively, caused by deploying new components in EC d. PUE_d is the power usage effectiveness value and y_d is the energy unit price for EC d. S_d is 1 if EC d is selected, otherwise 0.

Equations (4)–(7) formulate Y'_d and (12)–(16) formulate Y''_d. For the sake of notational simplicity, we skip the index of ECs in these equations.

$$Y'_d = \sum_{i \in N_d} SerEn_i \cdot S_i. \tag{4}$$

$$SerEn_i = \sum_{k \in M} E^i_{inc,k} \cdot L_{k,i}. \tag{5}$$

$$E^i_{inc,q} = (E^i_{wp} + E^i_{idle}) \cdot S^i_{Slp} + E^i_k. \tag{6}$$

$$E^i_k = (E^i_{max} - E^i_{idle}) \cdot \frac{C_k}{C^i_{max}}. \tag{7}$$

Subject to the following constraints:

$$L_{k,i} \le S_i. \qquad \forall i \in N_d, k \in M, \tag{8}$$

$$\sum_{i \in N_d} L_{k,i} = 1. \qquad \forall k \in M, \tag{9}$$

$$\sum_{k \in M} V_k \cdot L_{k,i} \le wN_i. \qquad \forall i \in N_d. \tag{10}$$

In (4), $SerEn_i$ is the incremental energy of running new components on CN i. S_i is 1 if at least one new component is deployed to CN i, otherwise 0. In (5), $E^i_{inc,k}$ is the incremental energy of running component k on CN i. $L_{k,i}$ is 1 if component k is allocated on CN i.

Equation (6) formulates $E^i_{inc,k}$ from (5), and where E^i_{wp} is the energy needed by CN i to go from sleep to active mode and E^i_{idle} is the energy consumption of CN i if idle (i.e., active, with

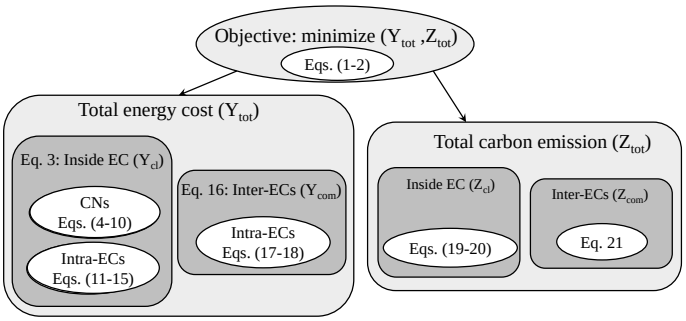


Fig. 2. Problem formulation diagram for the application allocation part

zero load). S^i_{Slp} is 1 if CN i is in sleep mode (0 if in active mode) and E^i_k is the additional energy consumption of running component k on CN i. If CN i is in sleep mode and receives the first component, it needs to spend energy E^i_{wp} to go from sleep to active mode. If active but idle, CN i consumes constant energy of E^i_{idle}; component k adds E^i_k to it. As the first component lets the CN wake from sleep mode, the resulting energy consumption is $E^i_{wp} + E^i_{idle} + E^i_k$. But for components added to an already active CN, the increase in energy is only E^i_k. To compute E^i_k, we use the following formula derived from [31] and [39]:

$$E^i = E^i_{idle} + \sum_{q=1}^{|M|} (E^i_q \cdot L_{q,i}) \tag{11}$$

In (7), E^i_{max} is the energy consumption of CN i with full load, C_k is the CPU load of component k and C^i_{max} is the CPU processing capacity of CN i.

The constraint, mentioned in (8), ensures that a component can be assigned only to a selected CN. Equation (9) guarantees that each component is assigned to exactly one CN and (10) guarantees that the total load of the components assigned to a CN does not exceed its capacity. Recall that V_k is the vector of requested resources of component k and wN_i is the current capacity of CN i.

After formulating Y'_d (incremental energy consumption of CNs in EC d), (12)–(16) formulate Y''_d (incremental network energy consumption in EC d). Y''_d is computed based on incremental network energy stemming from the additional traffic between each pair of CNs in EC d for running the new applications:

$$Y''_d = \sum_{i \in N_d} \sum_{\substack{i' \in N_d, \\ i \neq i'}} \sum_{t=1}^{\delta_{i,i'}} W^t_{i,i'}. \tag{12}$$

Here, $\delta_{i,i'}$ is the number of exchanged packets between CNs i and i', and $W^t_{i,i'}$ is the incremental energy of network elements between CNs i and i' for transferring the t^{th} packet. Note that in (12), only the selected CNs will be considered automatically, because when there is no traffic between CNs i and i', then $\delta_{i,i'} = 0$. $\delta_{i,i'}$ is computed based on the characteristics of the components allocated on CNs i and i' and on the traffic matrix:

$$\delta_{i,i'} = \sum_{k \in M} \sum_{k' \in M} L_{k,i} \cdot L_{k',i'} \cdot tr_{k,k'}. \tag{13}$$

$L_{k,i}$ is 1 if component k is allocated on CN i (otherwise=0) and $tr_{k,k'}$ is the number of packets between components k and k'. $W^t_{i,i'}$ is computed as follows:

$$W^t_{i,i'} = \sum_{B \in \lambda_{i,i',t}} E^B_{\text{inc}}. \tag{14}$$

Here, $\lambda_{i,i',t}$ denotes the path between CNs i and i' to which the t^{th} packet is assigned. E^B_{inc} is the incremental energy consumption of network element B for servicing a packet. The incremental energy consumption of network element B is computed analogously to that of CNs (see (6)):

$$E^B_{\text{inc}} = (E^B_{\text{wp}} + E^B_{\text{idle}}) \cdot N^B_{\text{Slp}} + E^B. \tag{15}$$

In (15), E^B is computed as indicated in [37].

$$E^B = E^B_p + E^B_{\text{S\& F}} L, \tag{16}$$

where E^B_p (i.e., per-packet processing energy) and $E^B_{\text{S\&F}}$ (i.e., per-byte store-and-forward energy) are constants for a given switch or router configuration, and L is the packet length.

Costs incurred by inter-cloud communication: Y_{com} is the incremental energy of the network to transfer data between different ECs while running the newly requested applications.

$$Y_{\text{com}} = \sum_{d,d' \in D} Y_{d,d'} \cdot y_{d,d'} \cdot S_d \cdot S_{d'}. \tag{17}$$

where $y_{d,d'}$ is the energy unit price for communication between ECs d and d'. $Y_{d,d'}$ is the incremental energy between ECs d and d' and is formulated in (18).

$$Y_{d,d'} = \sum_{t=1}^{\delta'_{d,d'}} \sum_{B \in \lambda'_{d,d'}} E^B_{\text{inc},t}. \tag{18}$$

$\delta'_{d,d'}$ is the number of exchanged packets and $\lambda'_{d,d'}$ is the set of network elements between ECs d and d'.

$$\delta'_{d,d'} = \sum_{i \in d} \sum_{i' \in d'} \sum_{k \in M} \sum_{k' \in M} L_{k,i} \cdot L_{k',i'} \cdot tr_{k,k'}. \tag{19}$$

where $L_{q,i}$ is 1 if AC_q is allocated on CN_i of EC_j (otherwise=0) and $L_{w,i'}$ is 1 if AC_w is allocated on $\text{CN}_{i'}$ of $EC_{j'}$ (otherwise=0). $tr_{q,w}$ is the number of packets between AC_q and AC_w.

3.2.2 Carbon emission formulation (Z_{tot})

Recall that Z_{tot} is the sum of Z_{cl} and Z_{com}, which are computed as follows.

Intra-EC carbon emission: Z_{cl} is the carbon emission caused by incremental energy in the selected ECs (CNs and intra-EC

networks) to run the requests, computed as:

$$Z_{cl} = \sum_{d \in D} PUE_d \cdot (Y'_d + Y''_d) \cdot CE_d \cdot S_d. \quad (20)$$

Recall that Y'_d and Y''_d (formulated in (4) and (12)) are the incremental server (CN) energy and network energy, respectively, in a selected EC d. S_d is 1 if EC d is selected, and 0 otherwise. CE_d is the average carbon emission rate (in g/kW) of the energy sources of EC d. It is computed as follows [4]:

$$CE_d = \frac{\sum_{k=1}^{\ell} CE_d^k \cdot r^k}{\sum_{k=1}^{\ell} CE_d^k}. \quad (21)$$

where CE_d^k and r^k denote the electricity generated by energy source k and its carbon emission rate, respectively.

Inter-EC carbon emission: Z_{com} is the amount of incremental carbon emission resulting from data transfer between the selected ECs:

$$Z_{com} = \sum_{d,d' \in D} Y_{d,d'} \cdot CCE_{d,d'} \cdot S_d \cdot S_{d'}. \quad (22)$$

where $CCE_{d,d'}$ is the average carbon emission rate for communication between EC d and d'. S_d is 1 if EC d is selected (otherwise 0). $Y_{d,d'}$ was defined in (18).

3.3 AC consolidation (migration) problem

Given a set M' of ACs running on CNs of EC d, our goal is to reduce the total energy consumption of running those ACs in d using migrations. That is, given a number of ACs (already allocated on CNs) with their sizes and traffic matrix as an input, we aim to find a new feasible placement for ACs allocated on under-utilized CNs, minimizing: (1) the energy spent to run the ACs (by consolidating ACs to reduce the number of active CNs), (2) the total network overhead (by improving placement of ACs to reduce inter-AC traffic and the number of active switches), and (3) the overhead of AC migrations (by selecting a closer destination for migration in order to reduce network energy/traffic and using already activated switches to have minimum number of active switches).

AC consolidation should obtain maximum energy saving using appropriate AC migrations while consuming minimum possible energy for the migrations. The AC consolidation problem can be formulated as follows:

$$\begin{cases} \text{maximize } (\vartheta - \vartheta' - \vartheta''), \\ \vartheta = \vartheta_s + \vartheta_n, \text{ and} \\ \vartheta' = \vartheta'_s + \vartheta'_n \end{cases} \quad (23)$$

where ϑ and ϑ' denote the energy consumption in EC d for running the ACs before and after AC consolidation, respectively. ϑ'' is the amount of energy consumed by the migrations.

ϑ is composed of the energy consumption of running the ACs on CNs (ϑ_s) and the energy consumption of the communication among them (ϑ_n). ϑ_s is computed as follows:

$$\vartheta_s = \sum_{i \in N_d} S_i \cdot \sum_{q \in M'} E_q^i. \quad (24)$$

Here, S_i is 1 if at least one AC is allocated on CN i, and E_q^i is the energy consumption of running AC q on CN i. ϑ_s is computed analogously to Y'_d (see (4)–(7)). The main difference is that ϑ_s contains the energy consumption of all allocated ACs on EC d, while Y'_d considers only the energy consumption needed for the new requests.

ϑ_n is the network energy consumption for the communication among the already allocated ACs:

$$\vartheta_n = \sum_{i \in N_d} \sum_{\substack{i' \in N_d \\ i \neq i'}} \sum_{t=1}^{\delta_{i,i'}} \sum_{\phi=1}^{H_{i,i'}} \alpha_{i,i',\phi}^t \cdot \sum_{B \in \lambda_{i,i',\phi}} E_{inc}^B. \quad (25)$$

$H_{i,i'}$ is the number of available paths between CN i and CN i'. $\alpha_{i,i',\phi}^t$ is 1 if the ϕ^{th} path is selected. ϑ_n is computed analogously to Y''_d (see (12)–(16)). The main difference is that ϑ_n includes the communication energy consumption of all already allocated ACs on EC d, while Y''_d considers only the communication energy consumption for the new requests.

ϑ' is calculated in the same way as ϑ, but using the new allocation after the migrations have been effectuated.

ϑ'' is the energy needed for the migration of the ACs selected by the AC consolidation algorithm.

$$\vartheta'' = \sum_{q \in M'} S_q \cdot \vartheta''_{q,(i,i')}. \quad (26)$$

where S_q is 1 if AC_q is migrated, otherwise 0. $\vartheta''_{q,(i,i')}$ is the energy needed to migrate AC_q from its source CN i to destination CN i':

$$\vartheta''_{q,(i,i')} = \sum_{t=1}^{T_q} \sum_{\phi=1}^{H_{i,i'}} \beta_{i,i',\phi}^t \cdot \sum_{B \in \lambda_{i,i',\phi}} E_{inc}^B. \quad (27)$$

T_q is the number of needed packets to transfer AC_q. T_q is computed based on the size of AC_q and packet size L (i.e., $\frac{AC_q}{L}$). $H_{i,i'}$ is the number of available paths between CN i and destination CN i'. $\beta_{i,i',\phi}$ is 1 if the ϕ^{th} path is selected.

4 Dynamic energy cost and carbon emission-efficient application placement method (DECA)

To describe DECA, first we introduce its general three-phase mechanism. Then, we present the DECA logical architecture and describe in detail each of its components.

4.1 General mechanism

DECA consists of three main phases: (1) Pre-allocation, (2) Allocation, and (3) Placement improvement.

Phase 1—Pre-allocation. Based on the requested application components, the components' traffic matrix, the available ECs and CNs, this phase tentatively determines the best resources considering cost and carbon emission. This is done in two steps (see Algorithm 1). Step 1 pre-selects ECs and pre-distributes the components to them simultaneously (joint EC selection and component distribution). Step 2 pre-chooses CNs in each pre-selected EC and pre-allocates components on them simultaneously (joint CN selection and component placement). In both steps, we first create candidate subgraphs and then select the best subgraph in terms of overall energy cost and carbon emission.

If a component is distributed to EC d in Step 1, then it is tried to be allocated on a compute node within EC d in Step 2. This will be normally successful since the decisions in Step 1 ensure that the total capacity of an EC is sufficient for the total demand of components mapped to that EC. However, this is only a necessary but not sufficient condition for being able to place the set of ACs

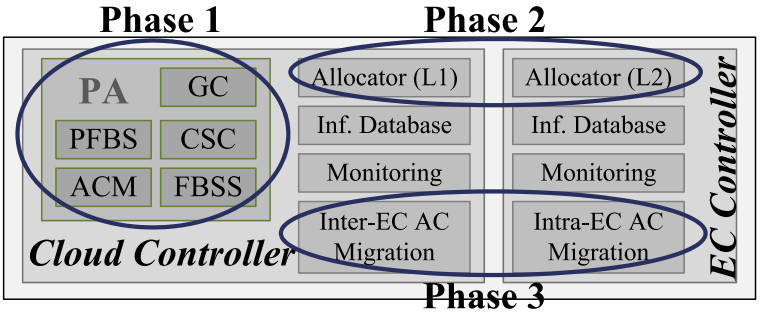


Fig. 3. Logical architecture of DECA

in this EC. It may turn out in Step 2 that an AC cannot be allocated in the intended EC. In this case, the AC and its related ACs are re-allocated to another EC.

Based on the pre-allocation observation, this phase finds the best resources for the requested ACs and offers them for allocation in Phase 2.

Phase 2—Allocation. Phase 2 actually allocates the requested ACs on the resources finally selected in Phase 1.

Phase 3—Placement improvement. Phase 3 is in charge of managing already allocated ACs. It has two main objectives: (1) energy saving: it includes inter-EC and intra-EC AC migration methods and continually minimizes energy consumption of each EC to optimize energy cost and carbon emission, (2) SLA violation prevention: by migrating ACs from overloaded CNs, it prevents SLA violation.

4.2 DECA architecture

As mentioned in Section 3.1, we assume a cloud controller and EC controllers in a hierarchical distributed EC architecture. Note that the cloud controller is a logical component which might be implemented in a physically distributed way enabling both load balancing and fault tolerance, but this is beyond the scope of the paper. In each time slot, the cloud controller selects appropriate resources (ECs, CNs and paths) for the received AC requests and distributes the ACs to the chosen ECs. Inside each EC, the EC controller places the received AC requests on the chosen CNs. Moreover, in each time period T, the cloud controller runs inter-EC AC migration and, right after that, each EC controller uses intra-EC AC migrations to continually optimize cost and carbon emission of the EC and also to prevent SLA violations (CN overloading).

To offer a better view about the DECA mechanism, we mapped the three phases of DECA on its logical architecture (see Fig. 3). Cloud and EC controller include some modules which are described next.

4.2.1 Cloud controller modules

Pre-Allocator (PA). This module (see Algorithm 1) reads general cloud information from the Information database, creates a complete graph of ECs and complete graph of CNs inside each EC, calls the appropriate sub-modules to pre-allocate the ACs, and finally sends the determined placement information to the Allocator (Level 1 and 2) modules. The PA module includes several sub-modules that we will describe as follows:

Algorithm 1: Pre-Allocator($G_0, \{G_i\}, T, TR$) $\rightarrow$ (c, f, e[f])

```
(a, b) = CSC(G_0, M, TR);   /*a: subgraphs of ECs, b: ACs on them*/
c = FBSS (a,b);   /*c: the best subgraph of ECs */
foreach EC ∈ c do
    (d, e) = CSC(G_EC , b[c][EC] , TR);
    /*d: subgraph of CNs, e: ACs on them*/
    f = FBSS(d, e);   /*f: the best subgraph of CNs for EC*/
/*c: selected ECs, f: selected CNs inside ECs, e: way of placing ACs on CNs*/
```

Graph Creator (GC). The GC module creates a complete graph of ECs and complete graph of CNs inside each EC based on general cloud information in the Information database.

AC Mapper (ACM). The ACM module (see Algorithm 2) receives a candidate vertex v (which may represent an EC or a CN) with its current capacity and a set X of ACs with their traffic matrix TR as input. Starting from each AC $q \in X$, ACM determines a subset Y_q of X that fits on v. Y_q is grown greedily: in each step, the AC from $X \setminus Y_q$ is chosen that has the largest traffic with the ACs already in Y_q, and is added to Y_q if v still has sufficient capacity. After creating a subset starting from each AC q, ACM selects the subset with highest inter-AC traffic and maps it to v.

Algorithm 2: ACM(v, X, TR) $\rightarrow$ highest traffic Y_i

```
foreach AC_q ∈ X do
    Y_i = {};
    if v has enough capacity for AC_q then
        Add AC_q to Y_i;
        while v has enough capacity for Y_i and Y_i ≠ X do
            Let V ∈ X \ Y_i have the highest total traffic with ACs in Y_i;
            if v has enough capacity for Y_i ∪ {V} then
                Add V to Y_i;
/* return the Y_i with highest inter-AC traffic */
```

Candidate Subgraph Creator (CSC). As Algorithm 3 shows, this module receives a weighted graph $G = (V, E, wV, wE)$, a list of ACs M[ ] and their traffic information TR as input and returns $|V|$ subgraphs (i.e., S) with allocated ACs on them (i.e., S_v). The

aim of CSC is to determine for each $v_i \in V$ ($1 \leq i \leq |V|$) an induced subgraph $G'(v_i)$ with sufficient total capacity for hosting the ACs and optimized overall cost and carbon emission. $G'(v_i)$ is grown from $\{v_i\}$ as starting point by iteratively adding one vertex a time. The already selected vertices of $G'(v_i)$ and their allocated ACs are stored in arrays $S[i]$ and $S_v[i]$ respectively. In each step, CSC checks whether the selected vertices have sufficient total capacity. If this is the case, $G'(v_i)$ is finished. Otherwise, the PFBS module is called to select one more vertex for inclusion in S, and the cycle continues, until the total capacity of the selected vertices is sufficient. Then, $G'(v_i)$ is the subgraph induced by $S[i]$, with its allocated ACs stored in $S_v[i]$. This way, a subgraph is created for each vertex v_i as a starting point yielding altogether $|V|$ candidate subgraphs.

Trying each vertex as a starting point is important because the subgraph formed starting from v_i will often be biased towards vertices in the proximity of v_i; taking the best one of the candidate subgraphs helps to find a globally optimal one. In principle, it would also be possible to consider all subgraphs of G with sufficient total capacity. However, the number of all such subgraphs can be exponential, making this approach intractable in practice. In contrast, our method is a faster, polynomial-time heuristic.

Prediction and Fuzzy Sets-Based Selector (PFBS). Whenever CSC needs to add a new vertex (i.e., v_k) to a candidate subgraph $G'(v_i)$ being generated, it calls the PFBS module (recall that the subgraph $G'(v_i)$ formed starting from v_i). As Algorithm 4 shows, PFBS receives a graph G, a list of already selected vertices in subgraph $G'(v_i)$ (i.e., $S[i]$), a set of unallocated ACs (i.e., X), AC traffic information (i.e., TR), and a variable H as input. PFBS returns the most cost/carbon effective vertex $v_k \in V \setminus S$ and selected ACs on it (S_v) to be included in $G'(v_i)$. H = 1 means all vertices of G will be used for allocating the requested ACs and so we do a simple random vertex selection to save time. $V \setminus S$ are the vertices still available in G for selection. Selecting the best vertex based on a single metric (e.g., only cost or carbon emission) is relatively easy. But selecting the best vertex according to two, sometimes conflicting, metrics simultaneously is more challenging. This is where Fuzzy Sets [7] are helpful to trade off the two metrics and thus select the best vertex. Another challenge is that, to be fast, we must make local decisions on the next vertex; but ignoring the effect of later decisions could lead to poor results. For this reason, we propose a combination of the Fuzzy Sets technique and the A^* algorithm [6] to get the benefits of both appropriate decisions for multiple metrics (using fuzzy sets) and global decisions (using A^*) together. An example showing the benefits of using A^* is given in the Appendix.

Equations (28)–(38) detail our proposed method. PFBS uses a fuzzy set of all possible (EC or CN) candidates (i.e., $V \setminus S$). There is a membership function $c()$ for this set to map each candidate to a membership value in the range [0, 1]. For a candidate $v_k \in V \setminus S$, $c(v_k)$ is computed based on its cost and carbon emission.

Inspired by the A^* algorithm, $c(v_k)$ is made up of two functions: $g(v_k)$ is based on the immediate costs and carbon emission incurred by selecting the candidate v_k, while $h(v_k)$ is based on an estimate of the costs and carbon emission that will be incurred in the future if v_k is selected now. This estimation aspect of the A^* algorithm can give a global optimization view. It also helps to consider the capacity of each candidate in addition to cost and carbon emission. Hence, for a candidate $v_k \in V \setminus S$, the

Algorithm 3: CSC(G, M[ ], TR) → (S, S_v)

```
S[ ][ ] = { },  S_v[ ][ ][ ] = { },  copy members of array M[ ] into a set X;
T_s: Total requested of |X| ACs,  T_n: Total available capacity of G
H = 1 if all nodes of G will be used for placing the requested ACs; else H = |V|;
for i=1 to H do
    S[i][0] = v_i, j = 0, X' = { }, S_v[i][j][ ] ← ACM(v_i, X \ X', TR),
    X' = X' ∪ S_v[i][j], T_c = wv_i; /*T_c :Total current capacity,*/
    while T_c < T_s do
        j ← j+1;
        (v_new, S_v[i][j]) = PFBS(G, S[i], X \ X', TR, H );
        /*selects one more vertex and its ACs*/
        X' = X' ∪ S_v[i][j],  S[i][j] = v_new;
        T_c = T_c + vv_new; /*vv_new is resource vector of v_new */
    /*if not all ACs could be allocated*/
    if X ≠ X' then
        X'' = { }; /*Set of ACs to move to another EC*/
        foreach x' ∈ X' do
            add x' to X'';
            foreach x ∈ X do
                if x' has relation with x then
                    add x to X'';
                    if x is already pre-allocated then
                        remove x from its CN and EC;
                    else
                        /*x is also a member of set X'*/
                        remove x from X';
        /*allocate X'' to an appropriate EC*/
        (S', S'_v) = CSC(G \ S[i], X'', TR)
        S = S ∪ S',  S_v = S_v ∪ S'_v;
/*returns candidate subgraphs with allocated ACs*/
```

Algorithm 4: PFBS(G(V,E,wV,wE) → (selected, $S_v[v_k]$)

```
c_max = −1;
if H == 1 then
    select a v_k ∈ V \ S randomly where v_k^cpu < SLA_Threshold;
    S_v[v_i][j] ← ACM(wv_i, X, TR);
    return (v_k, S_v[v_k]);
foreach v_k ∈ V \ S do
    if v_k^cpu < SLA_Threshold then
        S_v[v_i][j] ← ACM(wv_k, X, TR);
        compute c(v_k) using (28)–(38);
        if c(v_k) > c_max then
            c_max = c(v_k), selected = v_k;
```

A^*-based membership value is

$$c(v_k) = 0.5 \cdot (g(v_k) + h(v_k)). \tag{28}$$

The algorithm selects the candidate with the largest $c(v_k)$ value. Here, $g(v_k)$ computes a membership value based on the increment in overall cost and carbon emission (for both server and network resources) incurred by selecting v_k:

$$g(v_k) = 1 - K_1(v_k) \cdot K_2(v_k). \tag{29}$$

where $K_1(v_k)$ and $K_2(v_k)$ are normalized values of cost and carbon emission (in range of [0,1]) respectively. Equation (29) ensures that maximizing $g(v_k)$ leads to low values for both cost and carbon emission. Equations (30) and (31) normalize cost and carbon emission respectively.

$$K_1(v_k) = \frac{S_{v_k} \cdot P_{v_k} + N_{v_k} \cdot P'_{v_k}}{S_{max} \cdot P_{max} + N_{max} \cdot P'_{max}}. \tag{30}$$

$$K_2(v_k) = \frac{S_{v_k} \cdot CE_{v_k} + N_{v_k} \cdot CE'_{v_k}}{S_{max} \cdot CE_{max} + N_{max} \cdot CE'_{max}}. \tag{31}$$

where S_{v_k} is the incremental energy of v_k caused by running the new ACs and S_{max} is the incremental energy of the candidate with the maximum incremental energy. P_{v_k} and P'_{v_k} (for network elements inside v_i) are the energy price in the location of v_i. However, for inter-EC network elements, we consider P'_{v_k} as average of the energy prices. P_{max} and P'_{max} are the maximum energy price among all locations. CE_{v_k} and CE'_{v_k} (for network elements inside v_k) are the carbon emission rate in location of v_k. For inter-EC network elements, CE'_{v_k} is the average of the carbon rates. CE_{max} and CE'_{max} are the maximum carbon rate among all locations. N_{v_k} is the incremental energy of network elements caused by adding the candidate v_k to the subgraph $G'(v_i)$:

$$N_{v_k} = \sum_{v_j \in S[i]} n_{v_k, v_j}. \tag{32}$$

where n_{v_k,v_j} is the incremental energy of transferring data from candidate v_k to the already selected v_j in $G'(v_i)$ (recall that $G'(v_i)$ is the subgraph that is grown from v_i and S[i] holds the current vertices of $G'(v_i)$). For EC selection, n_{v_k,v_j} is computed based on (18) whereas for CN selection, (14) is used. PFBS calls for each candidate v_k the ACM module to detect allocated ACs on v_k and then computes $\delta_{k,j}$ for CNs based on (13) and for ECs based on (19). Recall that (14) also selects the best path between two CNs.

$h(v_k)$ computes a membership value based on an estimate of the increment in overall cost and carbon emission caused by the vertices that we will have to select later on to accommodate all the M ACs.

$$h(v_k) = 1 - K'_1(v_k) \cdot K'_2(v_k). \tag{33}$$

where $K'_1(v_k)$ and $K'_2(v_k)$ are normalized values of the estimated cost and carbon emission (in range of [0,1]) respectively.

$$K'_1(v_k) = \frac{y \cdot S_a \cdot P_a + U \cdot N_a \cdot P'_a}{y \cdot S_{max} \cdot P_{max} + U \cdot N_{max} \cdot P'_{max}}. \tag{34}$$

$$K'_2(v_k) = \frac{y \cdot S_a \cdot CE_a + U \cdot N_a \cdot CE'_a}{y \cdot S_{max} \cdot CE_{max} + U \cdot N_{max} \cdot CE'_{max}}. \tag{35}$$

where y and U are the estimated number of vertices and edges (network paths) to be added later to the subgraph $G'(v_i)$, when allocating the remaining ACs. S_a is the estimated average and S_{max} the maximum possible incremental energy for a new vertex, N_a is the estimated average and N_{max} the maximum possible incremental energy of the network for the further edges. P_a and P'_a are the average, P_{max} and P'_{max} the maximum price of energy for vertices and edges, respectively.

To estimate U, recall that G is a complete graph, so that each new node added to a subgraph $G'(v_i)$ with z vertices will add z new edges. After adding v_k to the subgraph $G'(v_i)$ with z vertices, it will consist of $z+1$ vertices, so adding further vertices will lead to $z+1, z+2, \ldots$ new edges. Hence, if y further vertices will have to be selected after v_k, we have

$$U = \sum_{k=z+1}^{(z+1)+(y-1)} k = z \cdot y + \frac{y \cdot (y+1)}{2}. \tag{36}$$

The value of y can be calculated as follows:

$$y = \frac{|M| - ((\sum_{v_j \in S[i]} F(v_j)) + F(v_k))}{Av}. \tag{37}$$

where $|M|$ is the total number of newly requested ACs, $F(v_j)$ is the number of allocated ACs on vertex v_j of subgraph $G'(v_i)$, $\sum_{v_j \in S[i]} F(v_j)$ is the number of allocated ACs till now on $G'(v_i)$, $F(v_k)$ is the number of ACs that can be allocated if v_k is chosen next, and Av is the average capacity of all vertices.

It remains to estimate N_a, the average network energy consumption for the edges that will be added to the subgraph $G'(v_i)$ in subsequent steps. One possibility is to use the average network energy consumption among all CNs. This would be a good estimate if we sampled edges randomly. However, our algorithm is biased towards edges of lower energy, so that the overall average may be an overestimate. A better estimate is the average energy consumption of the edges that the algorithm has selected so far, i.e., the edges within the subgraph $G'(v_i)$ extended with v_k (denoted as Al). However, when selecting the second vertex, $G'(v_i)$ has only one vertex and no edge, so in this case, we use the average network energy consumption between the first vertex (i.e., v_i) and all other vertices:

$$N_a = \begin{cases} \frac{\sum_{v_j \in Al} \sum_{v_\ell \in Al \setminus \{v_j\}} n(v_j v_\ell)}{(z+1)z/2} & \text{if } z > 1 \\ \frac{\sum_{v_j \in V \setminus Al} n(v_i, v_j)}{N-1} & \text{if } Al = \{v_i\} \end{cases} \tag{38}$$

Putting all the pieces together, we get a fairly good estimate of the overall energy cost and carbon emission when selecting v_i. Based on these estimates, the algorithm can select the best v_i (see Algorithm 4).

Fuzzy Sets-based Best Subgraph Selector (FBSS). This module receives as input a set of subgraphs (i.e., S) along with a list of allocated ACs on their vertices (i.e., S_v). FBSS selects the most appropriate subgraph in terms of cost and carbon emission. While selecting the best subgraph based on only cost or carbon emission is easy, the simultaneous optimization of the two metrics is more challenging. We use fuzzy sets to solve this.

As Algorithm 5 shows, FBSS computes for each $G'(v_i)$ the overall cost ($Cost_i$) and carbon emission (Car_i) using (39)–(40). Recall that vertices of a subgraph $G'(v_i)$ stored in S[i] and also S_v[i] consists of the allocated ACs of $G'(v_i)$.

$$Cost_i = \sum_{v \in S[i]} \sum_{j \in S_v[i][v]} E^v_{inc,j} \cdot y_v + \sum_{v,v' \in S[i]} Y_{v,v'} \cdot y_{v,v'}. \tag{39}$$

$$Car_i = \sum_{v \in S[i]} \sum_{j \in S_v[i][v]} E^v_{inc,j} \cdot CE_v + \sum_{v,v' \in S[i]} Y_{v,v'} \cdot CE_{v,v'}. \tag{40}$$

where $E^v_{inc,j}$ (incremental energy of vertex v of $G'(v_i)$ caused by running AC j) is computed based on (6)–(7), $Y_{v,v'}$ (incremental

Algorithm 5: FBSS(S[ ][ ],S_v[ ][ ][ ]) → (selected)

```
F_max = −1, i=0;
while S[i] ≠ null do
    compute cost of S[i] using (39);
    compute carbon emission of S[i] using (40);
    compute F_i using (41);
    if F_i > F_max then
        F_max = F_i, selected = S[i];
    i = i+1;
/* returns the best subgraph */
```

network energy between vertices v and v′) for EC subgraphs is computed from (18) and for CN subgraphs from (12)–(16). y_v and $y_{v,v'}$ (for network elements inside an EC) are the energy price in location of the EC. For inter-EC network elements, we consider $y_{v,v'}$ as average of the energy prices. CE_v and $CE'_{v,v'}$ (for network elements inside an EC) are the carbon emission rate in location of that EC. For inter-EC network elements, $CE'_{v,v'}$ is the average of the carbon emission rates. As we have the list of allocated ACs for each vertex, the number of exchanged packets between two nodes is easily computed for CNs from (13) and for ECs from (19).

FBSS then computes membership value of S[i] using the membership function shown in (41).

$$F_i = 1 - \frac{Cost_i}{Cost_{max}} \cdot \frac{Car_i}{Car_{max}}. \quad (41)$$

where $Cost_{max}$ and Car_{max} are the maximum possible cost and carbon emission for running the new requests, respectively. Finally, the subgraph with highest membership value is selected.

Allocator (Level 1). Based on AC allocation information from the Pre-Allocator module, this module sends each AC request to the chosen EC controller.

Monitoring. Cooperating with EC controllers, this module continuously monitors the cloud environment and updates the Cloud Information Database.

Information Database. All information related to the distributed EC is stored in this module.

Inter-EC AC Migration. The energy price and carbon emission rate not only can be different from EC to EC (i.e., in different locations) but can also vary with time (e.g., even on an hour-to-hour basis). Therefore, the overall energy cost or carbon emission may be reduced by shifting portions of the ACs to ECs that currently offer better energy prices and/or carbon emission rates [25]. Because of the similar decision-making for inter-EC and intra-EC AC migration, we describe them together in Section 4.2.2.

4.2.2 EC controller modules

The EC controller receives AC requests along with target CNs from the cloud controller. The EC controller allocates the ACs on the selected CNs (Allocator level 2), monitors the CNs (EC Monitoring) and gathers information of the EC and sends it to the cloud controller. To keep optimizing cost and carbon emission while preventing SLA violations, a Migration module is also used.

Allocator (Level 2). This module executes the AC allocation, as determined by the cloud controller.

Monitoring. Cooperating with CN controllers, this module continuously monitors the EC and updates the Information Database.

Algorithm 6: AC Migration-Under-utilized CNs (S,D)

```
Round_min = ∞, m=1, copy sets S to S′ and D to D′
while m != |S′| do
    for i=1 to |S| do
        T_stay=0, T_mgr=0
        foreach AC_q on CN_i ∈ S do
            F_min = ∞
            foreach CN_i′ ∈ D do
                compute F_(AC_q,CN_i′)  (see (42))
                if F_(AC_q,CN_i′) < F_min then
                    F_min = F_(AC_q,CN_i′), Index[i][q]=CN_i′
            if Index[i][q] is full, remove it from Set D
            compute F_(AC_q,CN_i)  (see (42))
            T_stay=T_stay+F_(AC_q,CN_i), T_mgr=T_mgr+F_min
        if (T_stay + E_idle > T_mgr) then
            pre-migrate (map) each AC_q of CN_i to
              Index[i][q], Round[i]=Round[i]+T_mgr
        else
            Round[i] = Round[i]+T_stay + E_idle,
              Index[i][q]=CN_i
    do a circular shift for set S′
    if (Round_min > Round[i]) then
        Round_min=Round[i], copy array Index[][] to array
          Temp[][]
    copy sets S′ to S and D′ to D, m++
migrate based on array Temp information
```

Information Database. All information related to the EC is stored in this module.

Intra-EC AC Migration. The aim of this module is twofold: (i) to prevent SLA violations of over-utilized CNs, and (ii) to optimize energy consumption (and thus costs and carbon emission) by emptying and switching off under-utilized CNs.

Inspired by [20], we consider three thresholds: T_L (low), T_M (middle) and T_H (high). These thresholds divide a CN's possible workload situations into four states: under-utilized (under T_L), light (between T_L and T_M), moderate (between T_M and T_H), and over-utilized (above T_H).

The goal of this workload classification is to easily find source and destination CNs for AC migration as follows: (1) all ACs on an under-utilized CN i should be migrated to CNs with light workload; then, i can be switched to sleep mode to save energy; (2) to prevent SLA violations, some ACs on over-utilized CNs must be migrated to other CNs (usually to CNs with light workloads); (3) ACs on lightly or moderately loaded CNs are not migrated to avoid unnecessary migration cost [20], [40].

For over-utilized CNs, migrating some ACs is mandatory to prevent SLA violation, even if this increases energy consumption. In contrast, migration of ACs from under-utilized CNs should be carried out only if it improves the energy balance, i.e., the energy consumption of the migrations is less than the energy reduction achieved.

Algorithm 6 shows the proposed AC migration mechanism for under-utilized CNs. In each time period, first, all under-utilized CNs are put in a set S and potential migration destinations are collected in a set D. These are primarily the lightly-loaded CNs, but if there is no such CN, we select moderately-loaded CNs

that have the capacity of receiving one AC without being over-utilized. If there are neither lightly nor moderately-loaded CNs to be included in D, we select a subset of the under-utilized CNs (e.g., the 30% with lowest load) and move them from S to D.

The AC Migration Module then selects an under-utilized CN from S (CN_i) and the first AC of it (AC_q). We estimate the energy needed for running AC_q on each destination $\mathrm{CN}_{i'} \in D$ ($1 \leq i' \leq |D|$) (in case of migration) or on CN_i (no migration):

$$F_{(AC_q, CN_{i'})} = \begin{cases} E_q^{i'} + N_q^{i'} + M_{q,(i,i')}, & \text{if } CN_{i'} \neq CN_i \\ E_q^{i'} + N_q^{i'}, & \text{if } CN_{i'} = CN_i \end{cases} \quad (42)$$

Here, $E_q^{i'}$ is the energy consumption of running AC_q on $\mathrm{CN}_{i'}$ (computed based on (7)) and $M_{q,(i,i')}$ is the energy needed to migrate AC_q from CN_i to $\mathrm{CN}_{i'}$, computed based on (27). $N_q^{i'}$ is the energy needed for the communication between AC_q and related ACs, assuming that AC_q is mapped to $\mathrm{CN}_{i'}$. Similar to (25), $N_q^{i'}$ is computed as:

$$N_q^{i'} = \sum_{\substack{CN_{i''} \in N \\ CN_{i''} \neq CN_{i'}}} \sum_{t=1}^{\delta^q_{i',i''}} \sum_{\phi=1}^{H_{i',i''}} \alpha^t_{i',i'',\phi} \cdot \sum_{B \in \lambda_{i',i'',\phi}} E^B_{inc}. \quad (43)$$

where $\delta^q_{i',i''}$ is the number of exchanged packets between AC_q on $\mathrm{CN}_{i'}$ and its related ACs on $\mathrm{CN}_{i''}$. N is the set of CNs in the EC.

$$\delta^q_{i',i''} = \sum_{AC_w \in CN_{i''}} tr_{q,w}. \quad (44)$$

If AC_w has no communication with AC_q, then $\mathrm{tr}_{q,w} = 0$; hence, only ACs communicating with AC_q are considered in (44).

Using (42), the AC Migration Module determines two values: (1) Stay value: value of the cost function when AC_q remains on CN_i (i.e., $F(AC_q, CN_i)$); (2) Migrate value: minimum value of cost function among all destination candidates in D (denoted F_{min}). In addition, the index of the CN with minimum value is saved.

The AC Migration Module repeats the above procedure to get $F(AC_q, CN_i)$ and F_{min} for each AC on CN_i. Then, all Stay values are summed to get T_{stay} and all Migrate values are summed to get T_{mgr}, leading to:

$$D_{mgr} = (T_{stay} + E^{CN_i}_{idle}) - T_{mgr}. \quad (45)$$

where $E^{CN_i}_{idle}$ is the idle energy consumption of CN_i. If $D_{mgr} > 0$, then running the ACs of CN_i on CN_i consumes more energy than migrating and running them on other CNs. In this case, all ACs of CN_i will be migrated to their already detected optimal destination CNs and CN_i is switched to sleep mode. But if $D_{mgr} \leq 0$, then migrating the ACs of CN_i would not save energy, so the AC Migration Module does not migrate any of them.

The AC Migration Module repeats this procedure for all under-utilized CNs (listed in set S) one by one. It should be noted that the actions of choosing destinations for underutilized CNs can affect each other (non-independent events [6]). For example, a CN that has become fully occupied by migrating ACs from underutilized CNs is not available anymore as destination for other underutilized CNs. Therefore, a different ordering of the source CNs may cause a different set of feasible destinations for each AC. In addition, the heterogeneity of ACs and of destination CN candidates makes it difficult to find a good ordering of the ACs. To cope with this issue, for a set S of under-utilized CNs, we make $|S|$ permutations with circle shifting their list by 1 to the left and run the algorithm again. Finally, the permutation which leads to the solution with the least energy consumption is selected.

Algorithm 7: AC Migration-Over-utilized CNs(S,D)

```
Set S: over-utilized CNs, Set D: destination candidates
foreach CN_i ∈ S do
    while CN_i is over-utilized do
        select an AC with highest membership degree
          (based on (46))
        select a CN with lowest cost from D; called CN_d
          (based on (47))
        after AC migration, if CN_d becomes full, remove
          it from D.
```

Now we come to the handling of over-utilized CNs. Note that our proposed AC allocation approach tries to avoid over-utilized CNs in the first place. However, to cope with applications with dynamically changing AC loads, the AC Migration Module supports an over-utilization avoidance mechanism.

First, the AC Migration Module puts all over-utilized CNs in a set S (see Algorithm 7). Then, the set D of potential destination CNs is built similarly as in the previous case.

Let CN_i be an over-utilized CN in S. To select an AC to migrate from CN_i, memory size and CPU utilization of ACs are both important: selecting ACs with low CPU load increases the number of necessary migrations, and migrating ACs with large memory size increases traffic and network energy consumption. Therefore, it is important to consider both metrics at the same time. To this end, we use a fuzzy set-based technique and compute a membership value for each ACq on CN_i:

$$F'(AC_q, CN_i) = \frac{c_q}{c_{max}} \cdot \left(1 - \frac{m_q}{m_{max}}\right). \quad (46)$$

where c_q and m_q are the CPU utilization and memory size of AC_q, respectively, and c_{max} and m_{max} are the maximum CPU utilization and maximum memory size, respectively. We select the AC_q with highest membership value, which we denote as AC_p.

To select a destination CN for the chosen AC_p, we compute a cost function for each candidate $\mathrm{CN}_{i'} \in D$, similarly as in (42):

$$F''(AC_p, CN_{i'}) = E_p^{i'} + N_p^{i'} + M_{p,(i,i')}. \quad (47)$$

$E_p^{i'}$ is the energy consumption of AC_p if it is run on $\mathrm{CN}_{i'}$ (computed based on (7)) and $M_{p,(i,i')}$ estimates the energy needed to migrate AC_p from CN_i to $\mathrm{CN}_{i'}$, computed based on (27). $N_p^{i'}$ is the energy of the communication between AC_p and its related ACs, assuming that AC_p is migrated to $\mathrm{CN}_{i'}$ (computed based on (43)).

We select the $\mathrm{CN}_d \in D$ for which $F''(AC_p, CN_d)$ is minimum. After migrating the selected AC_p to CN_d, if CN_d becomes full (i.e., migrating another AC to CN_d would make it over-utilized), then CN_d is removed from D.

If the CN_i is still over-utilized, the AC with the second highest membership value is selected, and this procedure continues until the CN is not over-utilized anymore. Afterwards, the second over-utilized CN is selected and the above procedure is repeated. This procedure continues until no over-utilized CN remains in EC_j.

Our inter-EC AC migration algorithm is similar to the above-mentioned AC migration algorithm for under-utilized CNs with considering the following two minor additional points: (1) sets S and D are from two different ECs. In order to detect possible

EC candidates for both source and destination of AC migration, we use a simple threshold-based technique. We consider a weight for each EC which shows its energy cost and carbon emission effectiveness in comparison to other ECs. Equation (48) computes the weight of an EC_j.

$$W_j = \frac{P_j}{P_{max}} + \frac{C_j}{C_{max}}. \quad (48)$$

where P_j, C_j, P_{max} and C_{max} are the energy price of EC_j, carbon emission rate of EC_j, highest energy price of all ECs and highest carbon rate of all ECs, respectively.

We then order all ECs based on their weights in a list. We first compare weights of the EC with highest weight with the EC with lowest weight, because AC migration between them has the most important effect on cost and carbon efficiency. In addition, we use a weight threshold. The aim of the threshold is to avoid costly migrations with questionable benefit. If the difference between weights of the highest-weight and lowest-weight ECs is less than the threshold, there is no need to do inter-EC AC migration. Otherwise, the inter-EC AC migration algorithm is called for the EC pair (the EC with highest weight is source and the EC with lowest weight is destination of migration). The AC migration continues until either the algorithm detects that it is more energy efficient if the ACs stay than if they migrate, or the capacity of the destination EC becomes exhausted. In the first situation, the highest weight EC is removed from the list of available ECs; in the second case, the lowest weight EC is removed from the list of candidates. After that, the comparison is repeated between the ECs with highest and lowest weight. Again, if the difference between their weights is higher than the threshold, the inter-EC AC migration algorithm is called, and so on. This module is carried out right before the migration module.

5 Performance Evaluation

To evaluate DECA, we considered a distributed EC system with 10 ECs and implemented three different scenarios using CloudSim [42], a simulator developed in Java [43]. Scenario I was designed for evaluating the AC placement algorithm of DECA showing the obtained trade off between carbon emission and energy cost while optimizing both at the same time. Scenario II emphasizes on evaluating the AC migration part of DECA for under-utilized CNs, and Scenario III for the evaluation of the migration algorithm of DECA for over-utilized CNs. Scenarios II and III validate the DECA migration phase ability in still decreasing the energy consumption after the initial placement.

Based on information from the US Energy Information Administration [45, Table.5.6.A], we consider energy price is in range [4, 20] cents/kWh and for each EC was randomly selected between 4 and 20. For the inter-EC network, the energy price was considered as the average (12 cent/kWh). The PUE value was considered in the range [1.56, 2.1] based on [16]. We considered six energy sources with different carbon emission rates from [4] (Nuclear: 15, Coal: 968, Gas: 440, Oil: 890, Hydro: 13.5 and Wind: 22.5 g/kWh), and assumed five different combinations with an average of 100, 200, 300, 400 and 500 g/kWh, respectively. We selected one of them randomly for each EC. We used real energy models for routers, switches and servers (CNs) from [37], [44]. The available capacity of each CN is between 10-15 slots. The traffic matrix of the ACs generates randomly. The randomly set parameters remain fixed across the runs of all tested algorithms to ensure comparability of the results.

TABLE 3
Scenario I details

EC number	10
Total free capacity	1000 to 2000 slots
EC free capacity	100 to 200 slots
AC requests	100, 200, and 300

The experiments were conducted on an Intel Core i9-9880H 2.30 GHz computer, with 32 GB of memory. Each execution run simulated one hour of traffic and completed in less than 1 minute for Scenario I, less than 6 minutes for Scenario II and less than 2 minutes for Scenario III. For each scenario, we also computed the average results of up to ten executions, i.e., less than 100 minutes of total simulation time (hence, validating the feasibility of our proposal, both in terms of computational and time constraints).

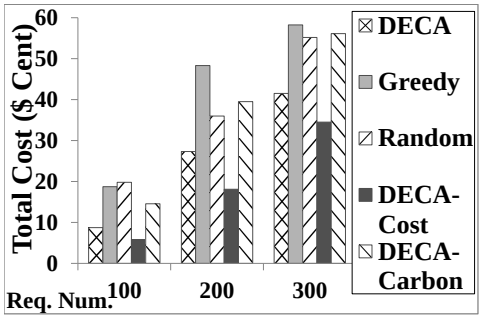
5.1 Scenario I

For Scenario I, the free capacity of the distributed EC system was chosen randomly, between 1,000 and 2,000 slots, in each run. This total capacity was divided among the 10 ECs (the free capacity of each EC was between 100 and 200 slots). Three different sets of requests with 100, 200, and 300 ACs were considered. Table 3 lists the details in Scenario I. In order to simplify our implementation, similar to [11], we describe our scenarios using the simplification of slots (i.e., each basic vector resource unit is represented by one slot). Nevertheless, our proposal can be implemented without using slots, as well.

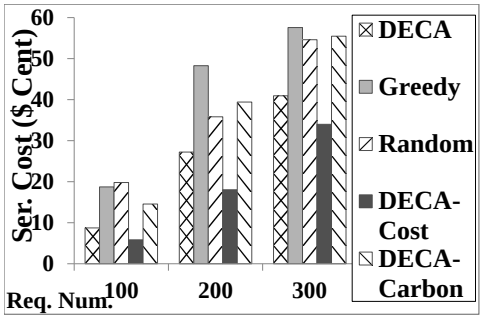
We compare the performance of DECA against DECA-Cost, DECA-Carbon, Random and Greedy resource allocation algorithms. DECA-Cost is a version of DECA which only considers cost optimization and DECA-Carbon only considers carbon footprint optimization. Comparing DECA to these special versions can show how DECA manages to find a trade-off between the two optimization goals. The Random algorithm starts by selecting a vertex (EC or CN) randomly and placing as many ACs as possible in this vertex. If not all ACs could be allocated in the selected vertex, then a further vertex is selected, again randomly, to place the remaining ACs. This process is repeated until all requested ACs are placed. The Greedy algorithm selects a vertex with maximum free capacity and allocates as many ACs from the request as possible in this vertex. If further ACs are necessary, then it selects from the remaining vertices again the one with maximum free capacity. This process continues until all ACs are placed [10], [14].

Figures 4, 5 and 6 show the simulation results (each run simulated 1 hour, i.e., that the energy cost and carbon emission values are for 1 hour). In particular, Figs. 4(a) and 5(a) show how DECA could successfully make a joint cost-carbon emission optimization. DECA outperforms the Random and Greedy algorithms in both dimensions. It was clear that DECA-Cost leads to the lowest cost. However, the carbon emission of DECA-Cost is sometimes even worse than that of Random or Greedy. Similarly, DECA-Carbon is the best in carbon efficiency but performs poorly in cost efficiency. In general, DECA improves 45-115% in carbon emissions on DECA-Cost while incurring 20-60% higher costs. In comparison to DECA-Carbon, DECA improved total cost by 40-60% while increasing carbon emission only by 10-30%.

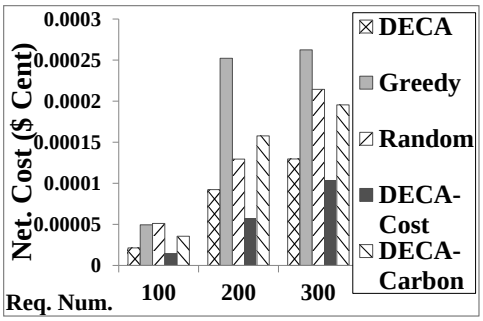
As shown in Fig. 6(a), Greedy always has the least number of selected ECs. Together with Figs. 4(a)–(d), this shows that only



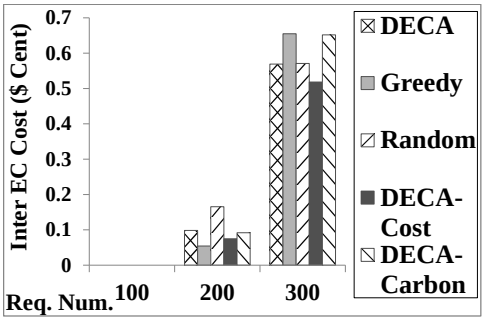
(a) total cost



(b) CN (server) cost

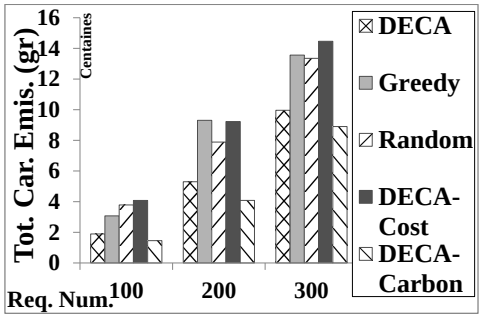


(c) intra-EC network cost

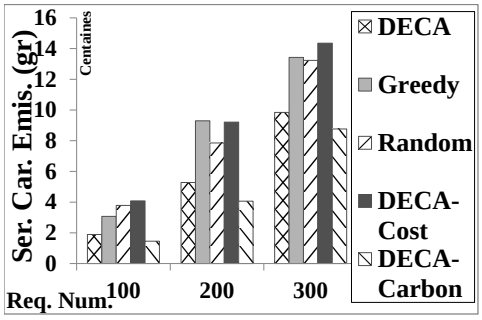


(d) inter-EC network cost

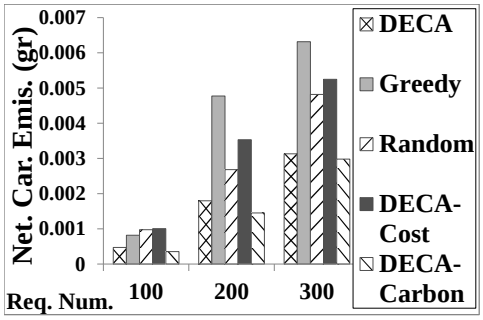
Fig. 4. Simulation results for Scenario I - Part I (cost)



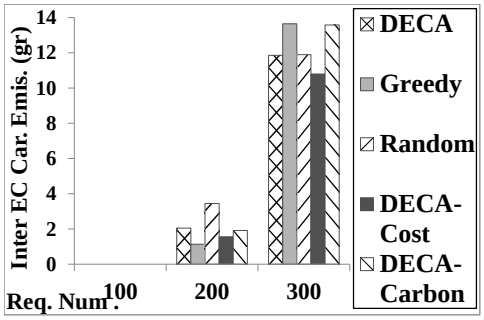
(a) total carbon footprint



(b) CN (server) carbon footprint



(c) intra-EC carbon footprint



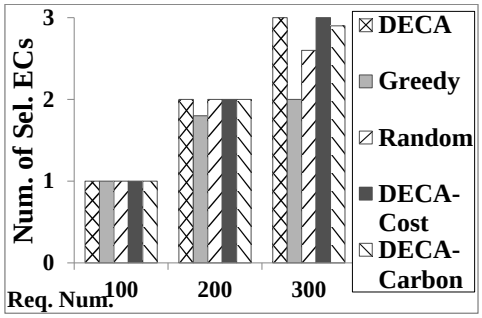
(d) inter-EC carbon footprint

Fig. 5. Simulation results for Scenario I - Part II (Carbon emission)

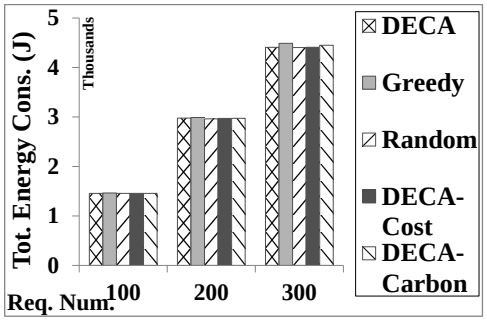
reducing the number of selected ECs (and CNs) does not lead to total cost optimum. In Fig. 6(b), there is no significant difference in energy consumption between the methods. Since there are still big differences in costs and carbon emissions, this shows the importance of taking into account the different energy sources and resulting variety of prices and emission rates of the ECs in a distributed cloud.

5.2 Scenario II

In Scenario II, we considered an EC with three different sizes: 100, 200 and 300 CNs. Each CN has a capacity between 10-15 slots. The traffic matrix of the ACs was generated randomly between 0 and 1,500 packets, where each packet has a size of 2 KB. Three different CPU loads and memory sizes are considered for the ACs (4, 8 and 12 GB). For migrations, we assume a page dirtying ratio of 0.2. The path length for each CN pair inside an EC was randomly chosen from 1 to 5 hops (switches). We used real energy



(a) number of selected ECs



(b) total energy consumption

Fig. 6. Simulation results for Scenario I - part III

TABLE 4
Scenario II details

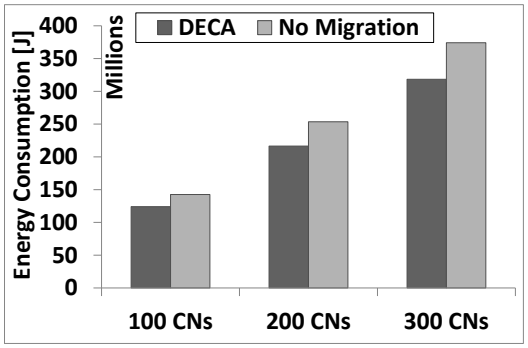
EC number	1
CN number	100, 200 and 300
Free capacity of a CN	10 to 15 slots
AC size	4, 8 and 12 GB
CN pair path length	1 to 5 hops
Packet size	2KB
Page dirtying ratio	0.2

models for routers and switches from [37] and for servers (CNs) from [44]. Table 4 lists the details associated to Scenario I.

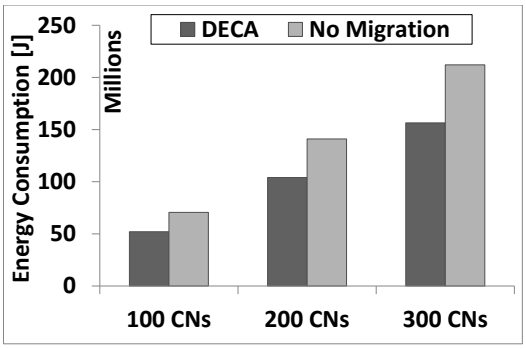
Scenario II is designed to evaluate the performance of our proposed AC migration algorithm for under-utilized CNs. The thresholds T_L, T_M and T_H are 0.2, 0.5 and 0.8, respectively. We assume that in the current placement of ACs on CNs, 30% of the CNs are under-utilized, 50% are lightly and 20% moderately loaded. To evaluate the effect of our approach accurately and individually, we assume now that there are no over-utilized CNs (over-utilized CNs will be considered in Scenario III).

Fig. 7(a) shows the effect of the AC migration algorithm for under-utilized CNs within DECA, compared to the situation without AC migrations. The results show that the AC migration algorithm of DECA for under-utilized CNs could reduce energy consumption. We would like to stress that the performance of our algorithm has a direct relation with the number of under-utilized CNs. Therefore, the resulting energy saving can be even better if the number of under-utilized CNs is higher.

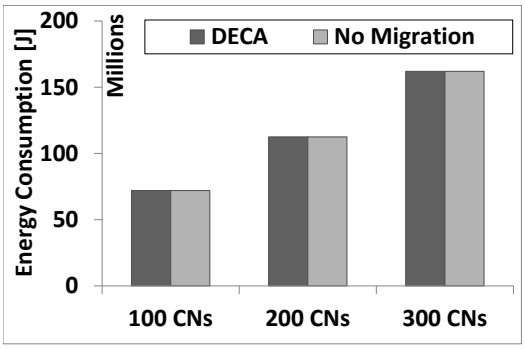
Fig. 7(b) shows that DECA significantly reduces computing energy consumption in comparison to no migrations by bringing under-utilized CNs to sleep mode. AC migration consumes energy, but this energy is negligible (see Figs. 8(a) and (b)) compared to the amount of energy saved by bringing under-utilized CNs to sleep (Fig. 7(b)). Note that the scale of Figs. 7(a)–(c) is



(a) total energy consumption



(b) compute energy consumption



(c) network energy consumption

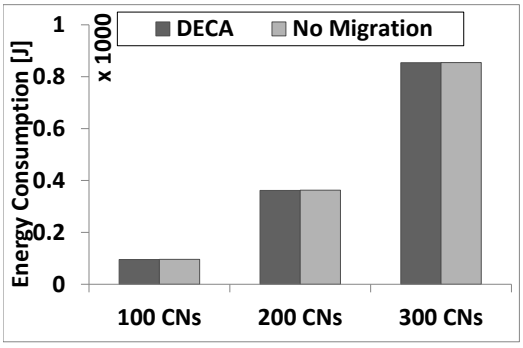
Fig. 7. Scenario II, effect of migrations from under-utilized CNs - Part I

TABLE 5
Energy saving [J] of different AC consolidation algorithms for different EC sizes

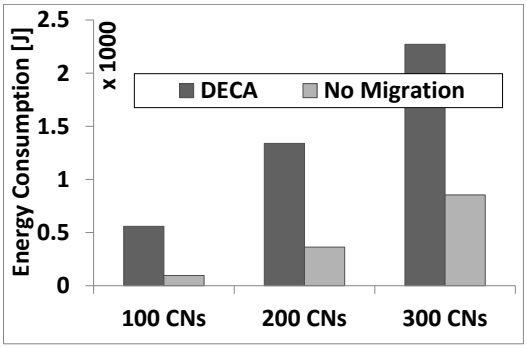
	100 CNs	200 CNs	300 CNs
DECA	18,604,070	37,031,702	55,652,790
Zhou [20]	18,555,649	36,953,253	55,539,514
Random	18,109,946	35,908,090	54,059,535

Millions and Figs. 8(a)–(b) scale is x1000. Fig. 8(a) shows that there is no visible difference in energy consumption of inter-AC communication, i.e., DECA did not increase inter-AC traffic.

Table 5 shows the energy saving of DECA for AC migration compared to the method of Zhou [20] and Random (i.e., selecting destinations randomly). Note that in contrast to the diagrams, here higher numbers are better. Energy saving is defined as $En_{bef} - (En_{aft} + En_{mgr})$, where En_{bef} and En_{aft} are the total energy consumption of compute and network resources without respectively with migrations, and En_{mgr} is the energy



(a) inter-AC traffic energy consumption



(b) energy consumption of inter-AC traffic and AC migrations

Fig. 8. Scenario II, effect of migrations from under-utilized CNs - Part II

TABLE 6
Comparing energy consumption for AC migration of DECA with other AC consolidation algorithms (Scenario II, under-utilized CNs)

	100 CNs	200 CNs	300 CNs
DECA	463.68	977.04	1418.16
Zhou [20]	458.88	976.8	1389.84
Random	477.84	966	1433.76

consumption of the migrations. As Table 5 shows, in all situations DECA saved more energy than the other methods. Tables 6, 7 and 8 provide more insight into the reasons.

Table 6 shows the energy consumption achieved with DECA for AC migration compared to other methods. As the objective of DECA is optimal energy saving, based on the situation, it may increase AC migration energy consumption so that it can reduce compute energy consumption more (and vice versa).

Table 7 shows energy consumption of compute resources achieved by the different methods. As can be seen, DECA has an advantage here. Although Zhou always selects the CN with the least increase in power consumption due to AC allocation, unlike DECA, it does not consider the order of selecting under-utilized CNs. Concerning energy consumption of network resources for inter-AC communication, Table 8 shows only marginal differences between the methods.

As a consequence, DECA aims to minimize the total energy consumption (i.e., compute, network and AC migration) rather than only minimizing compute energy consumption. DECA intelligently selects a CN which is not the one with the least increase in compute power consumption but incurs less energy for migration and inter-AC communication. In addition, because of the heterogeneity of ACs and destination CNs, DECA considers

TABLE 7
Compute energy consumption [J] of different AC consolidation algorithms for different EC sizes

	100 CNs	200 CNs	300 CNs
DECA	51,992,989	104,000,260	156,432,026
Zhou [20]	52,041,415	104,078,710	156,545,330
Random	52,487,098	105,123,882	158,025,266

TABLE 8
Network energy consumption [J] for inter-AC communication of different AC consolidation algorithms for different EC sizes

	100 CNs	200 CNs	300 CNs
DECA	95.22	361.73	853.76
Zhou [20]	95.07	362.12	853.58
Random	95.31	362.52	853.25

the order of selecting under-utilized CNs. All of them together lead to the best overall performance, as seen in Table 5.

5.3 Scenario III

We use the same configuration as in Scenario II, except for the differences mentioned below. Scenario III is designed to evaluate the performance of our proposed AC migration algorithm for over-utilized CNs. Now we assume that the placement of ACs on CNs is such that 30% of CNs are lightly-loaded, 50% moderately-loaded, and 20% over-utilized. To accurately and individually evaluate the effect of our algorithm for handling over-utilized CNs, we now assume that there is no under-utilized CN.

We compare the migration algorithm of DECA with three AC(VM) migration policies for over-utilized CNs: (1) MIMT, (2) MAMT, and (3) RCT [20]. While MIMT chooses the minimum number of ACs (i.e., selecting an AC with maximum CPU load) which must be migrated from a CN, MAMT selects the maximum number of ACs (i.e., selecting an AC with minimum CPU load) and RCT uses a random selection of ACs to decrease the CN's CPU utilization below a threshold. In addition to these three policies, to evaluate both the AC and CN selection parts of DECA accurately, we consider eight different combinations of CN and AC selection policies. Therefore, in total, we compare twelve policies:

- DECA: proposed CN selection / proposed AC selection (Proposed / Proposed),
- MIMT: select a CN with the least increase of power consumption / select an AC with maximum CPU load (Min. Compute / Max. CPU),
- MAMT: select a CN with the least increase of power consumption / select an AC with minimum CPU load (Min. Compute / Min. CPU),
- RCT: select a CN with the least increase of power consumption / a random selection of ACs (Min. Compute / Random),
- DECA CN selection / MIMT AC selection (Proposed / Max. CPU),
- DECA CN selection / MAMT AC selection (Proposed / Min. CPU),
- DECA CN selection / RCT AC selection (Proposed / Random),

- MIMT/MAMT/RTC CN selection / DECA AC selection (Min. Compute / Proposed),
- Random CN selection / DECA AC selection (Random / Proposed),
- Random CN selection / MIMT AC selection (Random / Max. CPU),
- Random CN selection / RTC AC selection (Random / Random),
- Random CN selection / MAMT AC selection (Random / Min. CPU).

Recall that all three MIMT, MAMT, and RCT migration policies use similar CN selection method (i.e., Min. Compute).

Figs. 9, 10 and 11 show energy consumption of the algorithms for different EC sizes (i.e., 100, 200 and 300 CNs respectively).

As these figures show, DECA outperforms all other methods for all 100, 200 and 300 CNs. "Min. Compute Proposed" (i.e., combination of the Min. Compute CN selection method with our proposed AC selection method) is the closest policy to DECA. However, in all situations, DECA outperforms the "Min. Compute Proposed" policy. Even for 300 CNs, DECA's performance (with 1102.79 J) is still better than "Min. Compute Proposed" policy (with 1124.69 J).

This is because both the CN selection and AC selection parts of DECA are efficient. While our CN selection considers energy efficiency in computing and network resources at the same time, the policy used in MIMT, MAMT, RTC (and many other methods, such as [22], [23]) only consider compute resource energy consumption (i.e., Min. Compute policy; choosing the CN with the least increase in power consumption due to AC allocation). Considering only compute energy consumption may lead to the selection of a destination CN far from the source and thus to higher energy consumption of AC migration.

Also, AC selection has a direct effect on energy consumption of AC migration. Although MIMT (i.e., selecting an AC with maximum CPU load) is a plausible method as it minimizes the number of AC migrations, our results show that MIMT may not be ideal if the memory sizes of ACs are different. For example, suppose that the CPU load of AC 1 is a bit higher than that of AC 2, while the memory size of AC 2 is much lower than that of AC 1. In this case, migrating AC 2 consumes less energy. Unlike MIMT which does not consider memory size of the ACs, our proposed AC selection policy takes into account both memory size and CPU load in selecting ACs.

Putting all the pieces together, such as considering both compute and network resources for CN selection and considering both CPU load and memory size for AC selection, we end up to a highly effective algorithm for migrating ACs from over-utilized CNs.

5.4 DECA overall discussion

This part consolidates the results obtained in the three scenarios.

Scenario I was designed for evaluating the AC placement algorithm (i.e., the initial placement) of DECA. Its results showed how DECA makes a trade off between carbon emission and energy cost while optimizing both at the same time. DECA combines a variant of the A* search algorithm [6] with a Fuzzy Sets technique [7]. Using powerful techniques (i.e., the A* search algorithm [6] with a Fuzzy Sets technique [7]), DECA could perform more effective optimization than traditional greedy heuristics.

Scenario II and III demonstrated the DECA migration phase ability in decreasing the energy consumption after the initial placement.

Scenario II was considered to evaluate the performance of our AC migration algorithm for under-utilized CNs. The first part of its results showed that we could significantly reduce computing energy consumption in comparison to no migrations by bringing under-utilized CNs to sleep mode. We found that AC migration energy consumption is negligible compared to the amount of energy saved by bringing under-utilized CNs to sleep. The second part of the Scenario II compared the energy saving of our AC migration algorithm for under-utilized CNs with the method of Zhou [20] and Random (i.e., selecting destinations randomly). In all situations, our AC migration algorithm saved more energy than the other methods. Because our AC migration algorithm aims to minimize the total energy consumption (i.e., compute, network and AC migration) rather than only minimizing compute energy consumption. It sometimes intelligently selects a CN which is not the one with the least increase in compute power consumption but incurs less energy for migration and inter-AC communication.

Scenario III was defined to evaluate the performance of our proposed AC migration algorithm for over-utilized CNs. Considering both compute and network resources for CN selection and also both CPU load and memory size for AC selection, we end up to a highly effective algorithm for migrating ACs from over-utilized CNs.

Putting all the pieces together, DECA showed promising results for two main parts: (i) determining the initial placement of newly deployed applications (i.e., Scenario I) and (ii) re-optimization of the placement of applications to react to workload changes (i.e., Scenarios II and III).

6 Conclusion

We have presented a Dynamic Energy cost and Carbon emission-efficient Application placement method (called DECA) for distributed Edge Clouds (ECs). It considers geographically varying energy prices and carbon emission rates as well as optimizing both network and compute resources at the same time. We showed that combining the prediction-based A* algorithm with a Fuzzy Sets technique can make an intelligent decision to optimize cost and carbon emission. We have proposed two AC migration algorithms and presented the effect of live AC migration algorithms on energy consumption in ECs. Three different scenarios have been considered to evaluate performance of DECA. Based on our experiments, we have seen that considering both compute and network resources for CN selection and both CPU load and AC memory size for AC selection can improve the energy cost and carbon efficiency. In this paper, DECA was targeting batch applications and force them to run at a cheaper time. As future work, we will deal with DECA's limitation for efficiently supporting more interactive environments in which it is not possible to control that all computing will happen within a slot of time. In addition, effectiveness of a SDN-based orchestration framework for managing live VM migration in order to minimize cost and carbon emission will be studied.

Acknowledgments

The work of Z. Á. Mann was partially supported by the European Union's Horizon 2020 research and innovation programme under grant 871525 (FogProtect). The authors also thank the anonymous reviewers for their constructive and insightful remarks that helped in improving this paper.

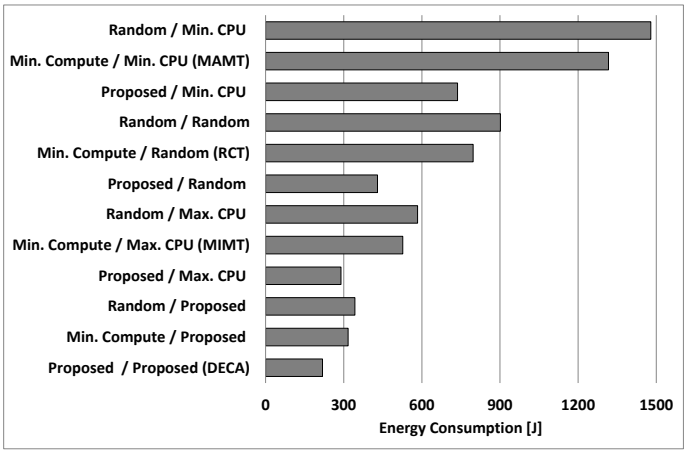


Fig. 9. Energy consumption achieved by different algorithms for intra-EC AC (VM) migration (over-utilized CNs) for 100 CNs (Scenario III). The first parameter shows CN selection method and the second one is AC selection (CN selection / AC selection)

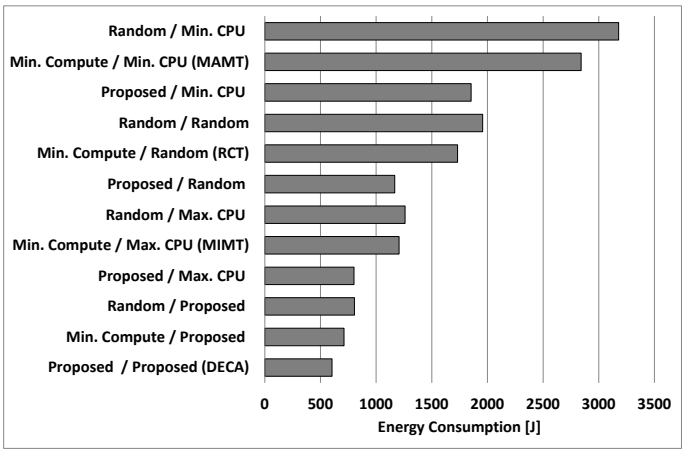


Fig. 10. Energy consumption achieved by different algorithms for intra-EC AC (VM) migration (over-utilized CNs) for 200 CNs (Scenario III). The first parameter shows CN selection method and the second one is AC selection (CN selection / AC selection)

REFERENCES

[1] J. Pan, and J. McElhannon, "Future edge cloud and edge computing for Internet of Things applications", IEEE J. IoT, vol. 5, no. 1, pp. 439–449, 2017.

[2] M. Chen, Y. Hao, L. Hu, M. S. Hossain, and A. Ghoneim, "Edge-CoCaCo: Toward joint optimization of computation, caching, and communication on edge cloud", IEEE Wirel. Commun, vol. 25, no. 3, pp. 21–27, 2018.

[3] P. Xiang Gao, A. R. Curtis, B. Wong, and S. Keshav, "It's Not Easy Being Green", ACM SIGCOMM, Finland, 2012.

[4] Z. Zhou, F. Liu, Y. Xu, R. Zou, H. Xu, J. C. S. Lui, and H. Jin, "Carbon-aware Load Balancing for Geo-distributed Cloud Services", IEEE MASCOTS, pp. 232–241, San Francisco, CA, 2013.

[5] S. Gosselin, F. Saliou, F. Bourgart, E. Le Rouzic, S. Le Masson, and A. Gati, "Energy Consumption of ICT Infrastructures: an Operator's Viewpoint", 38th ECOC Conf., pp. 1–3, Amsterdam, 2012.

[6] S. J. Russell, and P. Norvig, "Artificial Intelligence: A Modern Approach", Prentice Hall, 2010.

[7] L. Zadeh, "Fuzzy sets", Inform. Control, vol.8, pp. 338-353, 1965.

[8] Z.A. Mann, and M. Szabo, "Which is the best algorithm for virtual machine placement optimization?", Concurr. Comput. J. , vol. 29, no. 10, 2017.

[9] Z.A. Mann, "Allocation of virtual machines in cloud data centers – a survey of problem models and optimization algorithms", ACM Comput. Surv., vol. 48, no. 1, 2015.

[10] E. Ahvar, S. Ahvar, Z.A. Mann, N. Crespi, J. Garcia-Alfaro, and R. Glitho "CACEV: A Cost and Carbon Emission-Efficient Virtual Machine Placement Method for Green Distributed Clouds", IEEE SCC, pp. 275–282, San Francisco, USA, 2016.

[11] X. Li, J. Wu, S. Tang, and S. Lu, "Let's stay together: Towards traffic aware virtual machine placement in data centers", IEEE INFOCOM. pp. 1842–1850 Toronto, CA, 2014.

[12] A. Pahlevan, X. Qu, M. Zapater, and D. Atienza, "Integrating Heuristic and Machine-Learning Methods for Efficient Virtual Machine Allocation in Data Centers", IEEE Trans. on CAD, vol. 37, no. 8, pp. 1667–1680, 2018.

[13] K. You, B. Tang, and F. Ding, "Near-optimal virtual machine placement with product traffic pattern in data centers", IEEE ICC, pp. 3705–3709, 2013.

[14] M. Alicherry, and T.V. Lakshman, "Network aware resource allocation in distributed clouds", IEEE INFOCOM, pp. 963–971, 2012.

[15] E. Ahvar, S. Ahvar, N. Crespi, J. Garcia-Alfaro and Z.A. Mann, "NACER: a Network-Aware Cost-Efficient Resource allocation method for processing-intensive tasks in distributed clouds", IEEE NCA, pp. 90–97, USA, 2015.

[16] A. Khosravi, S. Kumar Garg, and R. Buyya, "Energy and Carbon-Efficient Placement of Virtual Machines in Distributed Cloud Data Centers", Euro-Par, pp. 317–328, 2013.

[17] A. Khosravi, L. L. H. Andrew, and R. Buyya, "Dynamic VM Placement Method for Minimizing Energy and Carbon Cost in Geographically Distributed Cloud Data Centers", IEEE Trans. Sustain. Comput., vol. 2, no. 2, 2017.

[18] C. Gu, C. Liu, J. Zhang, H. Huang, and X. Jia, "Green scheduling for cloud data centers using renewable resources", IEEE INFOCOM Ws., pp. 354–359, Hong Kong, 2015.

[19] N. Tziritas, C. Xu, T. Loukopoulos, S. Ullah Khan, and Z. Yu, "Application-aware workload consolidation to minimize both energy con-

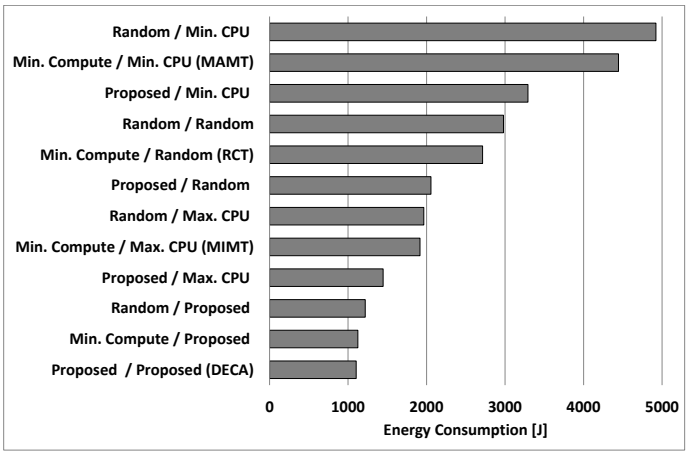


Fig. 11. Energy consumption achieved by different algorithms for intra-EC AC (VM) migration (over-utilized CNs) for 300 CNs (Scenario III). The first parameter shows CN selection method and the second one is AC selection (CN selection / AC selection)

sumption and network load in cloud environments", ICPP, pp. 449–457, France, 2013.

[20] Z. Zhou, Z. Hu, T. Song, and J. Yu, "A novel virtual machine deployment algorithm with energy efficiency in cloud computing", J. CENT SOUTH UNIV, Springer, vol. 22, no. 3, pp. 974–983, 2015.

[21] X. Zheng, Y. Cai, "Dynamic Virtual Machine Placement for Cloud Computing Environments", in Proc. 43th Int. Ws. ICPP, pp. 121–128, Minneapolis, 2014.

[22] A. Beloglazov, and R. Buyya, "Energy Efficient Allocation of Virtual Machines in Cloud Data Centers", CCGrid, pp. 577–578, Melbourne, 2010.

[23] A. Beloglazov, and R. Buyya, "Energy Efficient Resource Management in Virtualized Cloud Data Centers", CCGrid, pp. 826–831, Melbourne, 2010.

[24] X. Liu, Z. Zhan, J. D. Deng, Y. Li, T. Gu, and J. Zhang, "An Energy Efficient Ant Colony System for Virtual Machine Placement in Cloud Computing", IEEE Trans. Evol. Comput, vol. 22, no. 1, 2018.

[25] A. Forestiero, C. Mastroianni, M. Meo, G. Papuzzo, and M. Sheikhalishahi, "Hierarchical Approach for Efficient Workload Management in Geo-Distributed Data Centers", IEEE Trans. GCN, vol. 1, no. 1, 2017.

[26] X. Li, P. Garraghan, X. Jiang, Z. Wu and J. Xu, "Holistic Virtual Machine Scheduling in Cloud Datacenters towards Minimizing Total Energy", IEEE IEEE Trans. Parallel Distrib Syst, vol. 29, no. 6, pp. 1317–1331, 2018.

[27] M.H. Kabir, G.C. Shoja, S. Ganti, "VM Placement Algorithms for Hierarchical Cloud Infrastructure", CloudCom, Singapore, pp. 656–659, 2014.

[28] A. Singla, C. Hong, L. Popa, P. Brighten Godfrey, "Jellyfish: Networking Data Centers Randomly", in Proc. 9th USENIX Conf. NSDI, USA, 2012.

[29] M. Rahnamay-Naeini, S. Sen Baidya, E. Siavashi, and N. Ghani, "A Traffic and Resource-aware Energy-Saving Mechanism in Software Defined Networks", IEEE ICNC-SIREN, pp. 1–5, USA, 2016.

[30] S. Mustafa, K. Sattar, J. Shuja, S. Sarwar, T. Maqsood, S. A. Madani, and S. Guizani, "SLA-aware best fit decreasing techniques for work-load consolidation in clouds", IEEE Access, vol. 7, pp. 135256–135267,2019.

[31] C. Mobius, W. Dargie, and A Schill, "Power Consumption Estimation Models for Processors, Virtual Machines, and Servers", IEEE Trans. Parallel Distrib Syst, vol.25, no.6, pp. 1600–1614, 2014.

[32] N. Vasi, P. Bhurat, D. Novakovic, M. Canini, S. Shekhar,, and D. Kosti, "Identifying and Using Energy-Critical Paths", in Proc. 7th ACM Conf. CoNEXT, USA, 2011.

[33] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Banerjee, and N. McKeown, "ElasticTree: Saving Energy in Data Center Networks", NSDI, USA, 2010.

[34] Z.A. Mann, "Modeling the virtual machine allocation problem", MMSSE Conf., pp. 102–106, 2015.

[35] Z. Xu, and W. Liang, "Minimizing the Operational Cost of Data Centers via Geographical Electricity Price Diversity", IEEE Conf. on Cloud Comput., pp. 99–106, Santa Clara, 2013.

[36] I.S. Moreno, and J. Xu, "Customer-Aware Resource Overallocation to Improve Energy-Efficiency in Real-Time Cloud Computing Data Centers", IEEE Conf.SOCA, pp. 1–8, Irvine, USA, 2011.

[37] A. Vishwanath, K. Hinton, R.W.A. Ayre, and R.S. Tucker, "Modeling Energy Consumption in high-capacity routers and switches", IEEE J. Sel. Areas Commun., vol. 32, no.8, pp. 1524–1532, 2014.

[38] M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture", ACM SIGCOMM., USA, 2008.

[39] G. Warkozek, E. Drayer, V. Debusschere, and S. Bacha, "A new approach to model energy consumption of servers in Data Centers", IEEE Conf. ICIT, pp. 211–216, Athens, 2012.

[40] W. Voorsluys, J. Broberg, S. Venugopal, and R. Buyya, "Cost of Virtual Machine Live Migration in Clouds: A Performance Evaluation", Conf. CloudCom, Beijing, 2009.

[41] W. Fang, L. Xiangmin, S. Li, L. Chiaraviglio, and N. Xiong, "VMPlanner: Optimizing virtual machine placement and traffic flow routing to reduce network power costs in cloud data centers", Comput. Netw. J., vol. 57, no. 1, pp. 179–196, 2013.

[42] R.N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya, "CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms", Softw Pract Exp J., vol. 41, no. 1, pp. 23-50, 2011.

[43] Melbourne CLOUDS Lab, University of Melbourne. "CloudSim: A Framework for Modeling and Simulation of Cloud Computing Infrastructures and Services", Available On-Line: http://www.cloudbus.org/cloudsim/. Access : March 25, 2021.

[44] X. Zhang, J. Lu, and X. Qin, "BFEPM:Best Fit Energy Prediction Modeling Based on CPU Utilization", IEEE Conf. NAS, pp. 41–49, 2013.

[45] US Energy Information Administration. www.eia.gov/electricity/monthly/epm_table_grapher.cfm?t=epmt_5_6_a

[46] Eurostat Electricity Price Statistics . https://ec.europa.eu/eurostat/statistics-explained/index.php/Electricity_price_statistics

[47] M. Goudarzi, H. Wu, M. Palaniswami, and R. Buyya,"An Application Placement Technique for Concurrent IoT Applications in Edge and Fog Computing Environments", IEEE Trans Mob Comput, 2020.

[48] S. Pallewatta, V. Kostakos, and R. Buyya, "Microservices-based IoT Application Placement within Heterogeneous and Resource Constrained Fog Computing Environments", In Proc. 12th IEEE/ACM Conf. UCC, pp. 71–81, 2019.

[49] Y. Hu, C. d. Laat, and Z. Zhao, "Optimizing Service Placement for Microservice Architecture in Clouds", Appl. Sci. J., vol. 9, no. 21, 2019.

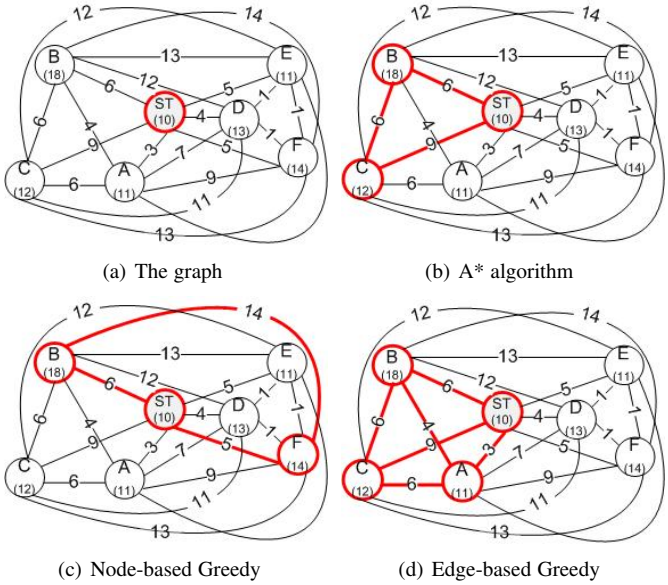
[50] H. O. Hassan, S. Azizi, and M. Shojafar, "Priority, Network and Energy-aware Placement of IoT-based Application Services in Fog-Cloud Environments", IET Commun, vol. 14, no. 13, 2020.

[51] P. Kayal and J. Liebeherr, "Autonomic Service Placement in Fog Computing", IEEE 20th WoWMoM, USA, 2019.

[52] S. Omer, S. Azizi, M. Shojafar, and R. Tafazolli, "A priority, power and traffic-aware virtual machine placement of IoT applications in cloud data centers", J. SYST ARCHITECT, vol. 115, 2021.

[53] S. S. Nabavi, S. Singh Gill, M, Xu, M, Masdari and P. Garraghan, "TRACTOR: Traffic-aware and power-efficient virtual machine placement in edge-cloud data centers using artificial bee colony optimization", Int. J. Commun. Syst., 2021.

[54] A. Ibrahim, M. Noshy, H. A. Ali and M. Badawy, "PAPSO: A Power-Aware VM Placement Technique Based on Particle Swarm Optimiza-



(a) The graph (b) A* algorithm

(c) Node-based Greedy (d) Edge-based Greedy

Fig. 12. An example of selecting subgraphs with different methods

tion," in IEEE Access, vol. 8, pp. 81747-81764, 2020, doi: 10.1109/ACCESS.2020.2990828.

APPENDIX

We show the benefit of the A* algorithm for application placement in ECs compared to other heuristics on a simple example. Fig. 12(a) shows a complete weighted graph on seven nodes with different capacities. The task is to allocate on the nodes a load of 40 units in total, with the objective of minimizing the total weights of edges (distance) between selected nodes. This example models a distributed EC environment with 7 DCs in which 40 VMs of equal size should be allocated. We consider three methods: allocating the VMs (1) greedily based on node capacity, (2) greedily based on edge weight, and (3) A* algorithm.

Node-based Greedy selects the nodes with largest available capacity, Edge-based Greedy selects nodes with shortest distance (based on edge weights) from already selected nodes. A* computes a cost value $c(v) = g(v) + h(v)$ for each candidate v, where $g(v)$ is the total distance of candidate v to already selected nodes and $h(v)$ is an estimate of total distance caused by adding the remaining nodes to allocate all 40 VMs. Finally, A* selects the candidate with lowest c value. The procedure of adding new nodes continues until all 40 VMs allocated.

Fig. 12(b)-(d) show the results of running the three methods on the graph of Fig. 12(a) with starting node ST. The Node-based Greedy method leads to the lowest number of selected nodes, as Fig. 12(c) shows. However, it selects nodes far from each other. In contrast, Edge-based Greedy selects nodes that are located as close as possible to each other; however, as it does not consider node capacity, it selects more nodes of lower capacity compared to Node-based Greedy (Fig. 12(d)). Therefore, as this example shows, both Node-based and Edge-based Greedy methods sometimes lead to poor results.

Unlike the two heuristics mentioned above, the A* considers both node capacity and distance at the same time, leading to better overall results (Fig. 12(b)). A* estimates for each candidate the future costs of selecting it in terms of how many additional nodes and edges will be necessary, and it selects the candidate with minimum total of already selected edge weights and estimated further edge weights. For example, if a low-capacity candidate is located in the proximity of the already selected nodes but — because of its low capacity — would lead to the selection of a higher number of nodes in the future to accommodate all requested VMs, it may be better to select a candidate that is farther away but offers higher capacity.