Efficient Algorithms for Multi-Component Application Placement in Mobile Edge Computing

Tayebeh Bahreini, *Student Member, IEEE,* and Daniel Grosu, *Senior Member, IEEE*

Abstract—In this paper, we address the Multi-Component Application Placement Problem (MCAPP) in Mobile Edge Computing (MEC) systems. We formulate this problem as a Mixed Integer Non-Linear Program (MINLP) with the objective of minimizing the total cost of running the applications. In our formulation, we take into account two important and challenging characteristics of MEC systems, the mobility of users and the network capabilities. We analyze the complexity of MCAPP and prove that it is NP-hard, that is, finding the optimal solution in reasonable amount of time is infeasible. We design two algorithms, one based on matching and local search and one based on a greedy approach, and evaluate their performance by conducting an extensive experimental analysis driven by two types of user mobility models, real-life mobility traces and random-walk. The results show that the proposed algorithms obtain near-optimal solutions and require small execution times for reasonably large problem instances.

Index Terms—Mobile edge computing, application placement, heuristic algorithm.

1 Introduction

The widespread usage of mobile devices generates an unprecedented amount of data that often requires real-time processing. This processing necessitates computational resources and storage capacity not available on mobile devices. Cloud computing is a promising technology that allows the mobile applications to offload their computations on cloud servers [1], [2], [3]. The main objective of offloading is to extend the battery life of mobile devices by executing heavy-computational components of the applications on remote servers. However, in cloud settings, computing services are usually far away from the end-user, and therefore, the communication between mobile devices and servers requires many network hops and results in high latencies. This is unfeasible for applications that require a very low latency or transmit large amounts of data [4].

In order to resolve this issue, several paradigms such as Cloudlet [5], Fog Computing [6], Follow Me Cloud [7], and Mobile Edge Computing (MEC) [8] have been recently proposed. The core idea of these paradigms is to offload a portion of data/computation to the edge of the network rather than offloading it to the cloud data-centers. Satyanarayanan et al. [5] proposed Cloudlet with the aim of bringing the cloud closer to the end user. A Cloudlet, also known as a micro-cloud, is a cluster of multi-core computers with high internal connectivity that is available to nearby mobile devices and provide computing, bandwidth, and storage services. Users access the Cloudlet servers via a local area network such as Wi-Fi. MEC [8] has been recently introduced to provide the required infrastructure for low latency computing services through running mobile applications at the edge of the network, where an edge can be any computing resources of the network. In MEC, the edge nodes are widely distributed in the network and available to all mobile users. On the other hand, being co-located with base stations, edge nodes have access to some additional information such as location and mobility of users.

One of the challenging issues in MEC systems is the resource scarcity. Compared to the cloud data centers, edge nodes have more restricted capacity. Therefore, it is not feasible to run a large size application on a single edge node. An efficient way to resolve this issue is to allow users to run the components of their applications on multiple edge nodes. Platforms such as Open Edge [9] and Open Fog [10] are developed for this purpose. They deploy virtualization techniques to share the resources of the edge nodes that are located in the same geographical region. In these platforms, finding an efficient placement for the components of an application on the multiple nodes is a major challenge.

Mobile users change their locations dynamically and the current assignment of the application to the edge nodes might not be the best in terms of the costs involved. In addition to the mobility of users, the resource availability and network conditions may also change dynamically. Therefore, in order to provide high quality services with the minimum costs, the application may need to migrate from one edge/core node to another, dynamically. The problem becomes more complex when an application has multiple components with heterogeneous requirements. The problem of assigning components of an application to the edge/core nodes such that the total cost of execution is minimized is called the *Multi-Component Application Placement Problem* (MCAPP). The components of a users' application can be run either on the core cloud, or on the edge of the network.

In MCAPP, an application can be represented as a graph in which the components of the application are the vertices,

• *T. Bahreini and D. Grosu are with the Department of Computer Science, Wayne State University, Detroit, MI 48202.*

E-mail: tayebeh.bahreini@wayne.edu, dgrosu@wayne.edu

and the edges between two vertices represent the communication between the corresponding components. Similarly, physical resources can be represented as a graph in which vertices are the computing resources (i.e., servers) and the edges between two vertices represent the communication links between the corresponding physical resources. Thus, MCAPP can be viewed as the problem of mapping the application graph onto the resource graph.

Our main contributions are as follows:

(1) Formulate the MCAPP problem as a Mixed Integer Non-Linear Program (MINLP). Our formulation of the problem departs from the existing work since it does not impose any restrictions on the topology of the graphs characterizing both the applications and the physical resources.
(2) Prove that MCAPP is NP-hard, which means that it is not solvable in polynomial time, unless $P = NP$.
(3) Design two efficient algorithms for solving MCAPP. Our goal is to design heuristic algorithms based on purely combinatorial techniques such as matching and local search, and to avoid the use of stochastic control-based approaches. The proposed algorithms have low complexity, and thus, add a negligible overhead to the execution of the applications.
(4) Evaluate the performance of the proposed algorithms by an extensive experimental analysis. The experiments are driven by two types of user mobility models, one derived from real-life mobility traces [11] and the other one based on the random-walk model [12]. We compare the performance of our algorithms against the optimal solution under the two types of mobility models. Our experimental results show that the proposed algorithms obtain near optimal solutions and require very small execution times.

The rest of the paper is organized as follows. In Section 2, we review the related work on the placement problem in cloud and edge computing. In Section 3, we introduce the multi-component application placement problem and present its MINLP formulation. In Section 4, we present the proposed heuristic algorithms. In Section 5, we illustrate the execution of the algorithms on a small instance of the problem. In Section 6, we present and analyze the experimental results. In Section 7, we conclude the paper and suggest possible directions for future research.

2 Related work

Edge nodes' proximity to users is a promising feature of MEC that can be exploited to improve the latency of the system. On the other hand, the mobility of users, the limited capacity of resources, and the dynamic nature of demands are critical issues that can affect the performance of the system. The existing techniques for application placement developed for cloud computing/data centers settings [13], [14] cannot be applied directly in the MEC setting because, when making the placement decisions, they do not consider the mobility of users and the differences in latency experienced by users at different locations. In this section, we review the existing literature on computation offloading and application placement problems in MEC addressing the above critical issues.

Many studies focused on computation offloading, where the computation requirements of applications and network conditions are taken into account to decide which tasks must be run locally and which tasks must be migrated to remote servers. In some studies [15], [16], [17], [18], [19], [20], energy consumption and computing latency have been considered as important performance metrics in optimizing computation offloading. The revenue of the service providers [21] and system's utility [22] were also considered as objectives in the computation offloading problem.

Several approaches for solving variants of the application placement problem in MEC have been proposed recently. Many of the dynamic application placement approaches formulated the problem as a sequential decision making problem in the framework of Markov Decision Processes (MDPs). Ksentini et al. [23] modeled the application/service migration problem considering the mobility of users in the Follow Me Cloud paradigm using MDPs [7]. In their formulation, they considered one dimensional mobility patterns. They implemented the value iteration algorithm in MATLAB to find the optimal application migration policy. Urgaonkar et al. [24] modeled the application placement problem as an MDP. To reduce the state space, they converted the problem into two independent MDP problems with separate state spaces and designed an online algorithm for the new problem that is provably cost-optimal. Wang et al. [25] presented a novel online algorithm for the application placement problem in the context of MEC. They modeled the problem as an MDP in which states are defined only based on the distance between user and servers.

Some researchers studied application placement problems for specific types of application graphs. Wang et al. [26] designed an online approximation algorithm for the placement problem in which both the application and the resource graphs are trees. The considered objective is to minimize the maximum weighted cost on each physical node and link of the system. Pei et al. [27] and Zou et al. [28] investigated the service chain embedding problem in MEC systems in which the application graph is a linear chain.

A few studies have focused on the placement of both servers and applications in a given network to improve the system performance [29], [30], [31], [32]. Balancing the load of servers and minimizing the delay of applications are two objectives that are considered in these studies. Several solutions have been proposed for the application placement problem that minimize the cost or the latency [33], [34], [35], [36]. These studies focused on the placement of the whole application on a single server and did not consider the possibility of assigning different components of an application to different edge servers.

However, the settings and objectives of the placement and offloading problems considered in the papers discussed above are different from those we consider in this paper and the algorithms do not directly apply to the multi-component application placement problem in MEC. They either considered a restricted topology for the application and resource graphs or considered different objectives. We aim at minimizing the total cost of the placement which includes the cost of running components on servers, the cost

of relocating a component from a server to another server, the communication cost between the user and a component, and the communication cost between components. We focus our efforts on designing efficient application placement algorithms that are suitable for implementation in real MEC systems.

3 Multi-component application placement problem

In this section, we formulate the *multi-component application placement problem (MCAPP)* in MEC systems. We consider a time slotted system, where T is the total number of time slots required to complete the execution of the application. The goal of the system is to determine the allocation of the components of the application in each time slot so that the total cost over T time slots is minimized. We formulate the problem for each time slot, where the relocation costs are determined by the allocation on the previous time slot. To make the formulation easier to understand and to avoid the use of an additional index to indicate the time slot for each variable, we present the problem only for one time slot.

In this formulation, we consider a two-dimensional grid area managed by an edge provider that periodically runs a resource manager. The system is composed of m servers $\{S_1, \ldots, S_m\}$ that are located at the edge of the network (e.g., at base stations). Note that in the rest of the paper, we use S_i and i interchangeably when referring to server S_i. We assume that the location of users may change from one time slot to another, where the location of a user is specified by its coordinates in a two-dimensional grid of cells.

The user requests to offload an application with n components $\{C_1, \ldots, C_n\}$. In the rest of the paper, we use C_j and j interchangeably when referring to component C_j. The processing requirement of component j is denoted by p_j. This represents the amount of component j's load that needs to be processed. We do not impose any restrictions on the communication between the components, any component can communicate with any other component of the application (i.e., the graph modeling the application is not restricted). We also assume that a server can communicate with any other server (e.g., via internet) incurring different costs for different servers. Here, the *objective* is to find an assignment of components to servers, such that the total placement cost of the application is minimized. The total placement cost is composed of four types of costs:

(i) γ_{ij}: *the cost of running component j on server i*. This cost is defined as the product of the cost of processing a unit load at server i and the amount of load that needs to be processed:

$$\gamma_{ij} = c_i \cdot p_j \tag{1}$$

(ii) $\rho_{ii'j}$: *the cost of relocating component j from server i to server i'*. In MEC, the locations of users may change during the execution of their applications. Also, the workload of the edge servers and other conditions of the network may vary from time to time. Therefore, it may be required to change the location where the components are running. The relocation cost is defined as follows:

$$\rho_{ii'j} = l_{ii'} \cdot q_j \cdot r \tag{2}$$

where $l_{ii'}$ is the distance between servers i and i', q_j is the size of component j that would migrate, and r is the cost of transferring one unit of data over one unit of distance. Since the managed area is a two-dimensional grid, the distance between servers is the Manhattan distance, that is, if server i is located in cell (x, y) and server i' is located in cell (x', y'), then the distance between the two servers is given by $l_{ii'} = |x - x'| + |y - y'|$.

(iii) δ_{ij}: *the communication cost between component j (assigned to server i) and the user*. In each time slot, data communication between components and the user may be required. This cost is defined as follows:

$$\delta_{ij} = d_i \cdot h_j \cdot r \tag{3}$$

where h_j is the size of data that must be transferred between component j and the user, and d_i is the distance between server i (that runs component j) and the user. The distance between the server and user is the Manhattan distance as defined in (ii) above.

(iv) $\tau_{ii'jj'}$: *the communication cost between components j and j' that are located on servers i and i', respectively*. Suppose that component j is located on server i and component j' is located on server i'. The communication cost between components is defined as follows:

$$\tau_{ii'jj'} = l_{ii'} \cdot g_{jj'} \cdot r \tag{4}$$

where $g_{jj'}$ is the size of data that must be transferred between component j and component j'.

TABLE 1: Notation

Notation	Description
m	Number of servers.
n	Number of components.
γ_{ij}	Cost of running component j on server i.
c_i	Cost of processing one unit of load on server i.
p_j	Processing requirement of component j.
$\rho_{ii'j}$	Cost of relocating component j from server i to server i'.
$l_{ii'}$	Distance between servers i and i'.
q_j	Size of component j.
r	Cost of transferring one unit of data over a unit of distance.
δ_{ij}	Communication cost between component j (assigned to server i) and the user.
d_i	Distance between server i and the user.
h_j	Size of data that needs to be transferred between component j and the user.
$\tau_{ii'j'}$	Communication cost between components j and j' that are located on servers i and i', respectively.
$g_{jj'}$	Size of data that needs to be transferred between components j and j'.
x_{ij}	Binary variable associated with the assignment of component j to server i.

Considering these, the total cost of the placement, which is the objective function of MCAPP, is given by,

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(\gamma_{ij}+\delta_{ij})\cdot x_{ij}+\sum_{i=1}^{m}\sum_{i'=1}^{m}\sum_{j=1}^{n}\rho_{ii'j}\cdot\bar{x}_{i'j}\cdot x_{ij}+\sum_{i=1}^{m}\sum_{i'=1}^{m}\sum_{j=1}^{n}\sum_{j'=1}^{n}\tau_{ii'jj'}\cdot x_{ij}\cdot x_{i'j'} \quad (5)$$

The decision variables x_{ij} are defined as follows: $x_{ij}=1$, if component j is assigned to server i in the current time slot; and 0 otherwise. Furthermore, $\bar{x}_{i'j}$ is not a decision variable but a parameter denoting the assignment of component j in the previous time slot, that is, $\bar{x}_{i'j}=1$ if component j was assigned to server i' in the previous time slot, and 0, otherwise. Note that in any time slot, the assignment of components in the previous time slot is known. Therefore, the objective function can be rewritten as,

$$\sum_{i=1}^{m}\sum_{j=1}^{n}(\omega_{ij}\cdot x_{ij}+\sum_{i'=1}^{m}\sum_{j'=1}^{n}\tau_{ii'jj'}\cdot x_{ij}\cdot x_{i'j'}) \quad (6)$$

Note that to make it easier to work with the objective function, we define ω_{ij} as $\omega_{ij}=\gamma_{ij}+\delta_{ij}+(\sum_{i'=1}^{m}\rho_{ii'j}\cdot\bar{x}_{i'j})$. In the rest of the paper, we call ω_{ij} the *server-component cost* and $\tau_{ii'jj'}$ the *inter-component cost*. In Table 1, we present the notation that is used throughout the paper. We now formulate MCAPP as a Mixed Integer Non-Linear Program (MINLP) and show that it is NP-hard. Then, we provide two heuristic algorithms to solve it.

MCAPP-MINLP:

$$\min \quad \sum_{i=1}^{m}\sum_{j=1}^{n}(\omega_{ij}\cdot x_{ij}+\sum_{i'=1}^{m}\sum_{j'=1}^{n}\tau_{ii'jj'}\cdot x_{ij}\cdot x_{i'j'}) \quad (7)$$

subject to:

$$\sum_{j=1}^{n}x_{ij}\leq 1 \quad i=1,\ldots,m \quad (8)$$

$$\sum_{i=1}^{m}x_{ij}=1 \quad j=1,\ldots,n \quad (9)$$

$$x_{ij}\in\{0,1\} \quad i=1,\ldots,m; \quad j=1,\ldots,n \quad (10)$$

According to the above formulation, the objective function of MCAPP is to minimize the total placement cost. The set of constraints (8) guarantees that each server is used by at most one component. The set of constraints (9) ensures that each component is assigned to exactly one server. The set of constraints (10) represents the integrality requirement for the decision variables. The optimal solution obtained by solving MCAPP-MINLP will be used in the experimental results section as a lower bound for the solution obtained by our proposed algorithms.

3.1 Complexity of MCAPP

We show that the decision version (MCAPP-D) of MCAPP is NP-complete. This implies that MCAPP is NP-hard. An instance of MCAPP-D is defined by: an application graph, a resource graph, server-component cost, ω_{ij}, component-component cost, $\tau_{ii'jj'}$, and a bound $B\in\mathbb{R}^{+}$. In the application graph, each vertex corresponds to a component and the weight of the edge between every two verices j and j' gives the total amount of inter-component communication between the corresponding components (i.e., $(g_{jj'}+g_{j'j})\cdot r$). In the resource graph, each vertex corresponds to a server and the weight of the edge between every two vertices i and i' gives the distance between the corresponding servers (i.e., $l_{ii'}$). The decision question is whether there is an assignment of components to servers such that the total cost of the assignment defined by Equation (6) does not exceed B.

Theorem 1. *MCAPP-D is NP-complete.*

Proof. We prove that MCAPP-D is NP-complete by showing that: (i) MCAPP-D belongs to NP, and, (ii) the Traveling Salesman Problem (TSP), a well-known NP-complete problem, can be reduced to this problem in polynomial time.

It is easy to show that MCAPP-D is in NP. We only need to guess an assignment from components to servers, and then, compute the total cost of the assignment (using Equation (6)) in polynomial time and check if it exceeds B or not.

For the second condition, we show that TSP is reduced to MCAPP-D in polynomial time. Let us define an arbitrary instance of TSP with bound L on the length of a tour, graph $G=(V,E)$, where V is the set of cities, E is the set of edges, and the weights $w_{ii'}$ for each edge (i,i') (i.e., the distance between cities i and i').

Now, we construct an instance of MCAPP-D, called P, based on G, such that the total cost of the assignment is less than B, if and only if we can find a tour in G with the total length less than L. Instance P has m servers and n components such that $m=n=\mid V\mid$. We also set $B=L$. In this instance, the resource graph, G', is the same as G and therefore, the distance between each pair of servers is the same as the distance between each pair of cities in G. For the application graph, we consider a ring graph in which the weight of the edge between each pair of adjacent vertices is 1 and the weight of edge between non-adjacent vertices is zero. Also, we assume that server-component computation costs, ω_{ij}, are zero.

We claim that G has a tour of total length less than L if and only if there is a solution for P, where the total cost is less than B. Let us consider a tour in G with total length less than L. We can consider this tour in G' and assign components to this tour in the order that they appear in the ring. Obviously, the total cost of this assignment is the same as the total length of the tour, and therefore, is less than B.

Conversely, suppose that there is an assignment from components to servers in P with total cost less than B. We assign these components to the corresponding cities in G and define a tour based on the order of the component in the ring. Clearly, the total length of this tour is the same as the total cost of the assignment, which does not exceed L. □

4 Algorithms for MCAPP

In the previous section, we showed that MCAPP is NP-hard. Therefore, it is not possible to find an optimal solution for it in polynomial time, unless $P=NP$. Thus, we need

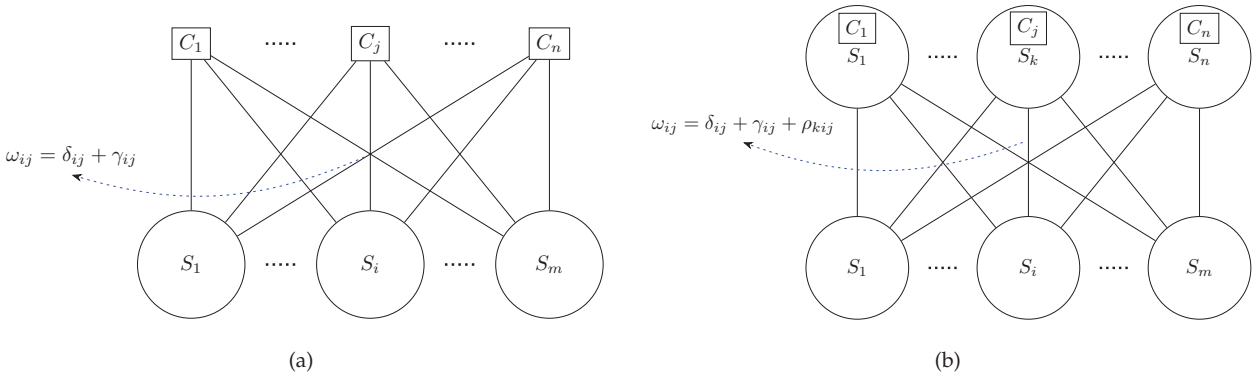

Fig. 1: Matching components to servers: (a) in the first time slot; (b) in other time slots.

to design efficient algorithms that obtain near-optimal solutions to MCAPP in polynomial time. For this purpose, we design two efficient heuristic algorithms MATCH-MCAPP and G-MCAPP. MATCH-MCAPP is an algorithm based on matching and local search techniques. This algorithm is very efficient for MEC systems with a relatively low number of servers and components, and applications with less intensive communication among components. In the absence of inter-component communications, MATCH-MCAPP obtains the optimal allocation. G-MCAPP is a greedy algorithm that is more suitable for MEC systems with a large number of servers and components, as well as for applications with intensive communication among components. Thus, in practice they can be deployed based on the types of instances that need to be solved. In the following, we describe the algorithms and discuss their properties.

4.1 MATCH-MCAPP Algorithm

We observe that the only factor that makes MCAPP NP-hard, is the existence of communication among components. Without this type of communication, the problem can be viewed as a matching problem (Fig. 1a) which is solvable in polynomial time. In the first time slot, the problem is to match components to servers, where each assignment of component j to server i has a specific cost given by:

$$\omega_{ij} = \delta_{ij} + \gamma_{ij} \tag{11}$$

The relocation cost is not considered in the first time slot, since $\rho_{ikj} = 0$. In the next time slots, the problem is to reassign components to servers taking users' location dynamics and other mentioned factors into account. In other words, the algorithm must decide whether a component stays on the current server or migrates to another one (Fig. 1b). The cost of assigning component j to server i must include the relocation cost and thus, it is given by:

$$\omega_{ij} = \delta_{ij} + \gamma_{ij} + \rho_{kij} \tag{12}$$

where k is the location of component j in the previous time slot.

Based on this fact, we design our first algorithm, called MATCH-MCAPP. This algorithm operates in two phases. In the first phase, it determines the best matching between components of the application and the servers without considering the communication requirements among the components (i.e., $\tau_{ii'jj'} = 0$). For this purpose, the algorithm uses the Hungarian algorithm [37] which finds the minimum cost assignment of the components to servers. In the second phase, the algorithm considers the communication requirements among the components and uses a local search procedure to improve the solution.

The Hungarian algorithm is a polynomial time algorithm that solves the assignment problem optimally. The algorithm has as input the weights ω_{ij} of the edges of the bipartite graph in which one partition is composed of vertices corresponding to the servers, and the other composed of vertices corresponding to the application components. The algorithm finds a perfect matching that gives the minimum computation cost of components to servers. Once the assignment is determined, the algorithm takes into account the communication costs between the components, $\tau_{ii'jj'}$ and performs a local search procedure that obtains the final solution to MCAPP.

MATCH-MCAPP is given in Algorithm 1. The algorithm is executed in each time slot for each application. The input to the algorithm consists of the cost parameters, ω_{ij}, and $\tau_{ii'jj'}$. To make it easy to describe the algorithm, we use the following notation: $\boldsymbol{\omega}$ is the array of server-component costs; and $\boldsymbol{\tau}$ is the array of inter-component costs. The values of these parameters are determined during the previous time slot and are used as input to the algorithm in the current time slot. The output of the algorithm is the assignment matrix $X = \{x_{ij}\}$, and the total cost of running the application on the assigned servers, $cost$.

In the first phase, the algorithm determines the optimal assignment of the components to servers by calling the function HUNGARIAN($\boldsymbol{\omega}$) (Line 1). This function implements a variant of the Hungarian algorithm and takes as input the cost $\boldsymbol{\omega}$ and returns the assignment as the vector $\boldsymbol{y}$; where $y_j = i$ if component j is assigned to server i. Since the Hungarian algorithm is well known we will not describe it here, but we refer the reader to Kuhn [37] .

Since the inter-component cost is not considered, the Hungarian algorithm is able to determine the optimal assignment of components to servers. This optimal assignment is not a solution for the MCAPP problem, it is an optimal assignment for the MCAPP with zero costs for the communication between components (i.e., $\tau_{ii'jj'} = 0$).

Algorithm 1 MATCH-MCAPP Algorithm

{Executed every time slot }
Input: $\boldsymbol{\omega}$: server-component costs
$\boldsymbol{\tau}$: inter-component costs
1: $\boldsymbol{y} \leftarrow$ HUNGARIAN($\boldsymbol{\omega}$)
2: $cost \leftarrow \sum_{j=1}^{n}(\omega_{y_j j} + \sum_{j'=1}^{n} \tau_{y_j y_{j'} j j'})$
3: $toVisit \leftarrow \{1, \ldots, n\}$
4: **while** $toVisit \neq \emptyset$ **do**
5: **for each** $j \in toVisit$ **do**
6: $\Lambda_j \leftarrow \sum_{j'=1}^{n} \tau_{y_j y_{j'} j j'}$
7: $b \leftarrow \text{argmax}_{j \in toVisit}\{\Lambda_j\}$
8: $toVisit \leftarrow toVisit \setminus \{b\}$
9: **for** $i = 1, \ldots, m$ **do**
10: $\boldsymbol{y} \leftarrow$ SWAP-COMPONENTS(y_b, i)
11: $new\_cost \leftarrow \sum_{j=1}^{n}(\omega_{y_j j} + \sum_{j'=1}^{n} \tau_{y_j y_{j'} j j'})$
12: **if** $new\_cost < cost$ **then**
13: $cost \leftarrow new\_cost$
14: **else**
15: $\boldsymbol{y} \leftarrow$ SWAP-COMPONENTS(i, y_b)
16: **for** $j = 1, \ldots, n$ **do**
17: $x_{y_j j} \leftarrow 1$
Output: $(X, cost)$

Algorithm 2 G-MCAPP Algorithm

{Executed every time slot }
Input: $\boldsymbol{\omega}$: server-component costs
$\boldsymbol{\tau}$: inter-component costs
1: $S \leftarrow \emptyset$
2: **for** $i = 1, \ldots, m$ **do**
3: **for** $j = 1, \ldots, n$ **do**
4: $\sigma_{ij} \leftarrow \omega_{ij}$
5: $S \leftarrow S \cup \{(i, j)\}$
6: **while** ($S \neq \emptyset$) **do**
7: $(i^*, j^*) \leftarrow \text{argmin}_{(i,j) \in S}\{\sigma_{ij}\}$
8: $x_{i^* j^*} \leftarrow 1$
9: **for each** $(i, j) \in S$ **do**
10: **if** $i = i^*$ **or** $j = j^*$ **then**
11: $S \leftarrow S \setminus \{(i, j)\}$
12: **for each** $(i, j) \in S$ **do**
13: $\sigma_{ij} \leftarrow \sigma_{ij} + \tau_{ii^* jj^*} + \tau_{i^* ij^* j}$
14: $cost \leftarrow \sum_{i=1}^{m}\sum_{j=1}^{n} \omega_{ij} \cdot x_{ij} + (\sum_{i'=1}^{m}\sum_{j'=1}^{n} x_{ij} \cdot x_{i'j'} \tau_{ii'jj'})$
Output: $(X, cost)$

Then, in the second phase, MATCH-MCAPP performs a local search that takes into account the cost of communication between components (Lines 2-15).

In the second phase, first, MATCH-MCAPP computes the cost of the current assignment determined by the Hungarian algorithm and also adds the inter-component costs to obtain the total cost (Line 2). Then, the algorithm defines a set, $toVisit$, and initializes it with the set of all components (Line 3). Next, it computes the total inter-component cost, Λ_j, of each component j in the $toVisit$ set (Lines 5-6). Then, in line 7, it determines the index of the *bottleneck component*, denoted by b. The bottleneck component is the component that has the maximum value for the total inter-component cost, Λ_j. The algorithm removes this component from the $toVisit$ set (Line 8). Thus, this component will not be selected as the bottleneck in the next iterations. After that, it executes a for loop (Lines 9-15) in which it tries to find a lower total cost assignment by swapping the component that is currently placed on server i with the bottleneck component. This is done by calling the function SWAP-COMPONENTS(y_b, i). This function swaps the components that are located at servers y_b and i, that is, assigns the bottleneck component to i and the component that is currently placed on server i to the server in which b resided. If there is no component on server i, then b is assigned to i and the server on which b resided is marked as available. The function outputs a new assignment vector $\boldsymbol{y}$. After this, in line 11, MATCH-MCAPP computes new_cost, the total cost of the system under the new assignment. If there is an improvement in the cost, it updates the total cost, $cost$, otherwise it restores the previous assignment by calling the SWAP-COMPONENTS function (Lines 12-15). The algorithm continues this procedure as long as there is an unvisited component. Then, it updates the assignment matrix X based on the assignment vector y (Lines 16-17).

We now investigate the time complexity of MATCH-MCAPP. The time complexity of the first phase, Hungarian algorithm, is $O(\max(m^3, n^3))$. Since, we assume that the number of servers is greater than the number of components, the time complexity of the Hungarian algorithm is $O(m^3)$. The most computational expensive section of the second phase consists of lines 4-15. The time complexity of the first part of this section (Lines 5-8) is $O(n^2)$ while that of the second part (Lines 9-15) is $O(mn^2)$. Furthermore, since each component is not chosen as the bottleneck more than once, these two parts are not executed more than n times. Thus, the time complexity of the second phase is $O(mn^3)$. Therefore, the time complexity of MATCH-MCAPP is $O(m^3 + mn^3)$.

4.2 G-MCAPP Algorithm

G-MCAPP is a greedy algorithm that finds the assignment of components to servers iteratively. To determine the assignment, the algorithm considers both the server-component and the inter-component costs simultaneously. The idea of G-MCAPP is to assign a component to a server in each iteration in such a way that the minimum cost is added to the total cost. For this purpose, the algorithm employs the *assignment cost* variable, σ_{ij}.

In the first iteration, since the assignment of none of the components has been determined yet, G-MCAPP only decides based on the server-component cost (i.e., $\sigma_{ij} = \omega_{ij}$). The algorithm selects a server-component pair (i^*, j^*) that has the minimum value of $\sigma_{i^* j^*}$. Then, for each unassigned server-component pair, (i, j), the algorithm updates the assignment cost by adding the inter-component cost between the previously selected component and the current component (i.e., $\sigma_{ij} = \omega_{ij} + \tau_{ii^* jj^*} + \tau_{i^* ij^* j}$). In the next iterations, G-MCAPP decides the assignment based on the updated costs and continues the procedure of selecting the server-component pair with the minimum assignment cost.

G-MCAPP is given in Algorithm 2. The input to the algorithm consists of: $\boldsymbol{\omega}$, the server-component cost matrix; and $\boldsymbol{\tau}$, the array of inter-component costs. In this algorithm, S is the set of all possible pairs of servers and components. For each pair (i, j), variable σ_{ij} is initialized to ω_{ij} (Lines 2-5). Then, iteratively, the algorithm finds the assignment of each component. First, the algorithm

selects a server-component pair (i^*, j^*) that has the minimum server-component cost and assigns component j^* to server i^* (Lines 7-8). Since each component is assigned to exactly one server and each server is used by at most one component, the algorithm removes all pairs that contain i^* or j^* from set S (Lines 9-11). Then, for each unassigned component j and server i, the algorithm updates the assignment cost by considering the inter-component communication between the component j^* and component j (i.e., $\sigma_{ij} \leftarrow \sigma_{ij} + \tau_{ii^*jj^*} + \tau_{i^*ij^*j}$) (Lines 12-13). Therefore, in the next iteration of the algorithm, the assignment costs are updated and the inter-component communication costs are considered. The algorithm continues this procedure until all the components are assigned. Finally, it determines the total cost based on the assignment of components (Line 14).

Now, we investigate the time complexity of G-MCAPP. The algorithm executes n iterations. The most time consuming part of each iteration consists of computing the minimum pair and updating the cost of the remaining pairs. In the first iteration, since there are $m \cdot n$ pairs, the time complexity of these operations is $O(mn)$. In the next iteration, the number of pairs reduces to $(m-1)(n-1)$. Generally, in the i-th iteration, the number of pairs is $O((m-i+1)(n-i+1))$. Therefore, the time complexity of G-MCAPP is $O(\sum_{i=1}^{n}(m-i+1)(n-i+1)) = O(mn^2)$.

5 An Illustrative Example

We provide a numerical example to show how our algorithms work. We consider an edge system consisting of three servers $\mathcal{S} = \{S_1, S_2, S_3\}$, and an application with three components $C = \{C_1, C_2, C_3\}$. Fig. 2a shows the user-server and server-server distances (i.e., d_i and $l_{ii'}$) in the previous time slot. In this figure, the weights on solid line edges are the server-server distances and the weights on the dashed line edges are the user-server distances. We assume that in the previous time slot, components C_1, C_2, and C_3 have been assigned to servers S_2, S_1, and S_3, respectively.

In the next time slot, as the user's location changes, the user-server distances change too (See Fig. 2b). Therefore, the algorithms may need to change the assignment. Fig. 2b and Fig. 2c show the new assignment of the components in the next slot obtained by MATCH-MCAPP and G-MCAPP, respectively. In these figures, we observe that MATCH-MCAPP changes the location of components C_1 and C_2 while G-MCAPP decides not to change the location of any component. In the rest of this section, we show how these two algorithms decide on their assignment. The values of the parameters are provided in Table 2. Based on these parameters, we obtain ω_{ij} given in Table 3.

TABLE 2: Example: The values of the cost parameters (r, data transmission cost rate; $g_{jj'}$, size of data transferred between components j and j'; h_j, size of data transferred between user and component j; p_j, processing requirement of component j; c_i, cost of processing of one unit load on server i).

Parameter	Value
r	1
$< g_{12}, g_{13} >$	$< 12, 15 >$
$< g_{21}, g_{23} >$	$< 13, 20 >$
$< g_{31}, g_{32} >$	$< 20, 30 >$
$< h_1, h_2, h_3 >$	$< 5, 10, 10 >$
$< q_1, q_2, q_3 >$	$< 4, 2, 2 >$
$< p_1, p_2, p_3 >$	$< 2, 3, 2 >$
$< c_1, c_2, c_3 >$	$< 5, 10, 12 >$

TABLE 3: Example: The values of the server-component costs, ω_{ij}.

i/j	1	2	3
1	72	115	122
2	32	66	62
3	51	76	54

5.1 MATCH-MCAPP

In the first phase of MATCH-MCAPP, the Hungarian algorithm is employed to determine the placement, which is C_1 to S_1, C_2 to S_2, and C_3 to S_3 (Fig. 3a). According to Equation (7), the total cost of this assignment is 757. In the second phase of MATCH-MCAPP, the local search, takes the inter-component communication ($\tau_{ii'jj'}$) into account. In each iteration of the local search, the total inter-communication cost, Λ_j, of each component j is computed and the component with the maximum value of Λ_j is selected as the bottleneck. In the first iteration, the total communication cost of each component is computed: $\Lambda_1 = 285$, $\Lambda_2 = 325$, $\Lambda_3 = 460$. Therefore, component C_3 is selected as the bottleneck. In the figures, the bottleneck component is represented by a solid black square. Then, the algorithm swaps the bottleneck with the component in the next server if it leads to a reduction in the total cost. In this case, the algorithm assigns C_3 to S_1 and C_1 to S_3, because it reduces the total cost from 757 to 714 (Fig. 3b). In the next iteration, the algorithm skips swapping the bottleneck with component C_2 because the total cost of this possible assignment is 758 which is greater than the total cost obtained from the previous assignment. Since all possible swaps for the current bottleneck have been tried, the algorithm starts the next iteration to select another bottleneck. In the next iteration, the total inter-component communication costs of the remaining components are calculated: $\Lambda_1 = 335$, $\Lambda_2 = 275$. Therefore, C_1 is selected as the bottleneck. The algorithm skips swapping server S_3 with S_1 since it will not reduce the total cost (Fig. 3d). In the next step, the algorithm again skips swapping S_3 and S_2, because it will not reduce the total cost (Fig. 3e).

In the next iteration of the algorithm, component C_2 is selected as the bottleneck. The algorithm swaps S_2 with S_1 because it reduces the total cost from 714 to 703 (Fig. 3f). The algorithm skips swapping S_1 with S_3 because it will not reduce the total cost (Fig. 3g) and it stops because all the components have been visited. Therefore, the total placement cost obtained by MATCH-MCAPP is 703.

5.2 G-MCAPP

Now, we show how G-MCAPP determines the placement of the components. The algorithm initializes the assignment cost between each pair of components and servers, σ_{ij}, based on the values of the server-component costs (i.e., $\sigma_{ij} = \omega_{ij}$). In each iteration, the algorithm selects a pair of server i^* and component j^* for which the value of $\sigma_{i^*j^*}$ is minimum and assigns component j^* to server i^*. Then, it updates the value of σ_{ij} for the unassinged pairs of

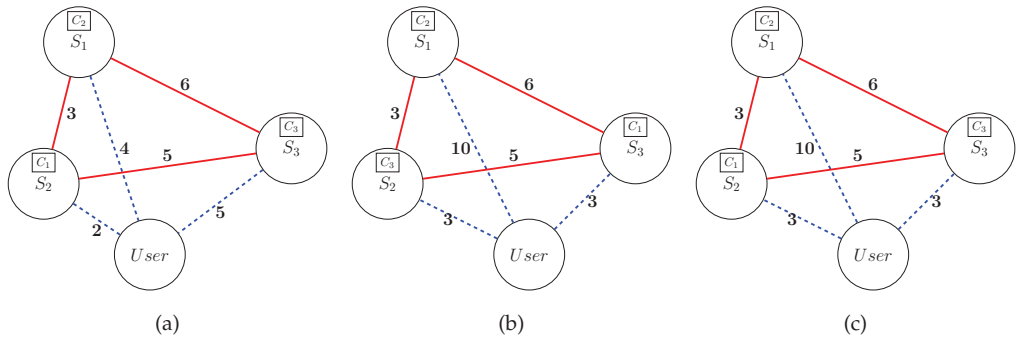

Fig. 2: Example: (a) The placement of components on servers in the previous slot; (b) The placement of components obtained by MATCH-MCAPP; (c) The placement of components obtained by G-MCAPP.

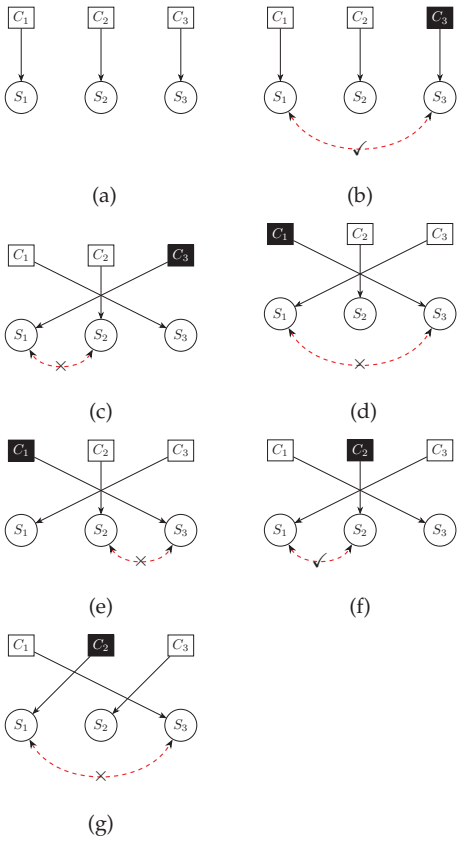

Fig. 3: Example: Second phase (local search) of MATCH-MCAPP on an instance with three components and three servers.

servers and components. Table 4 shows the values of the assignment costs in each iteration of G-MCAPP. The algorithm selects (C_1, S_2) with the cost of 32 as the pair with the minimum server-component cost and assigns C_1 to S_2. Then, it updates the assignment cost of each remaining pair (i, j) by adding the inter-component cost between C_1 and component C_j (i.e., $\sigma_{ij} = \omega_{ij} + \tau_{i2j1} + \tau_{2i1j}$). Therefore, for example the cost of assigning component C_2 to server S_1 is updated to $\sigma_{12} = \omega_{12} + \tau_{1221} + \tau_{2112}$. The values of the assignment costs obtained in the second iteration of the algorithm are given in the second row of Table 4. In this

TABLE 4: Example: Assignment costs, σ_{ij}, in each iteration of G-MCAPP

iteration	σ_{11}	σ_{12}	σ_{13}	σ_{21}	σ_{22}	σ_{23}	σ_{31}	σ_{32}	σ_{33}
1	72	115	122	**32**	66	62	51	76	54
2	–	190	227	–	–	–	–	201	**189**
3	–	**490**	–	–	–	–	–	–	–

table, the pairs that are not allowed to be selected (due to the Constraints (8) and (9)) are marked with " – ".

In the second iteration, the pair (C_3, S_3) with a cost of 189 is selected as the pair with the minimum cost and therefore, C_3 is assigned to S_3. Then, the assignment costs of the remaining pairs are updated (i.e., $\sigma_{ij} = \omega_{ij} + \tau_{i3j3} + \tau_{3i3j}$). In the last iteration, the algorithm assigns component C_2 to server S_1. Therefore, based on Equation (7), the total placement cost obtained by G-MCAPP is 765.

Comparing the results obtained by the two algorithms, the total cost obtained by G-MCAPP is 8.8% higher than that obtained by MATCH-MCAPP. In Section 6, we show that the quality of solutions obtained by MATCH-MCAPP is better than G-MCAPP for small-size problem instances, specifically, when the amount of inter-component communication is not high.

6 Experimental Results

We perform extensive experiments in order to investigate the properties of the proposed algorithms. We compare the performance of the algorithms against that of the optimal solution obtained by solving MCAPP-MINLP and that of another placement algorithm. In the following, we describe the experimental setup and analyze the experimental results.

6.1 Experimental Setup

Because the development of MEC is still in the early stages, there are no MEC workload traces that are publicly available. Therefore, for our experiments, we have to rely on synthetically generated instances for the MCAPP problem. In the following, we describe how we generate the problem instances that drive our simulation experiments and describe the experimental setup.

We consider a time slotted system in which the locations of users in the network may change from one time slot to another, but do not change during one time slot. To evaluate

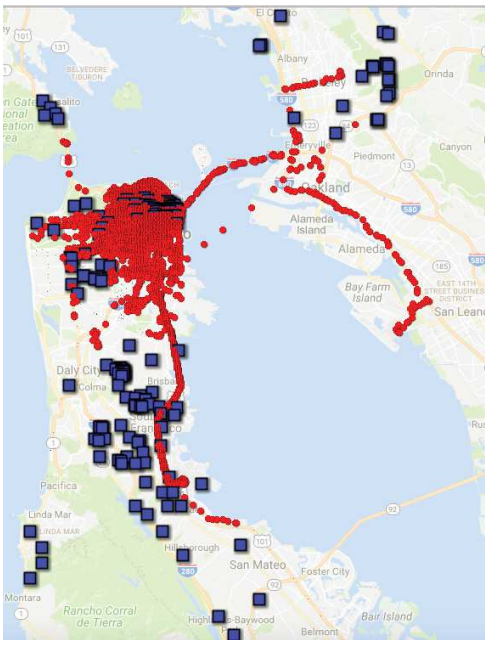

Fig. 4: Distribution of the edge servers (blue squares) and the frequent paths of the users (red dots))[Images generated using GPS Visualizer [38]].

the efficiency of the algorithms, we consider two different mobility models for the users: (i) Trace-Driven (TD), based on real-world mobility data [11], and (ii) Random Walk (RW) model [12].

For the trace-driven experiments, we use the CRAWDAD data set containing mobility traces of taxi cabs in San Francisco, CA [11]. The data set contains the GPS coordinates of about 500 taxi cabs collected over 30 days. We randomly choose the traces of 150 taxi cabs whose locations are updated every 10 seconds and use them as mobility traces for the users in our experiments. We also consider 200 edge servers that are co-located with 200 selected cell towers in the San Francisco area. The locations of these towers are obtained from antennasearch.com. In our experiments we do not consider towers with height less than 100 feet. Fig. 4 shows the distribution of towers in San Francisco and the most frequent paths that are used by the selected 150 taxis in the area. For the trace-driven experiments, we set the length of a time slot to 5 minutes. Every experiment is repeated ten times and each time, we select a taxi randomly from the data set and run the experiments.

For the second sets of experiments, those using the random walk mobility model, we assume that the mobility of users is based on the random walk model in a two-dimensional space. The users and servers are located within a two-dimensional grid of 50×50 cells. In fact, we consider the area of the San Francisco taxi traces as a 50×50 grid. Initially, a user can be in any cell of the grid network and its location is drawn randomly from a uniform distribution over the locations of the grid. In our setting, in every new time slot, a user can stay in its place or move into any of neighboring cells with equal probability. The servers are located within the same two-dimensional grid network and the coordinates of their positions are the same as those of servers we considered in the experiments with the trace-driven data set. The distance between servers and users is the Manhattan distance (as defined in Section 3).

TABLE 5: Simulation parameters

Parameter	Description	Distribution
c_i	Cost of processing one unit of load on server i.	$N(\mu_i, 0.2\mu_i)$, $\mu_i \sim U[1, 10]$
p_j	Processing requirement of component j.	$N(\mu_j, 0.2\mu_j)$, $\mu_j \sim U[0, 10]$
r	Data transmission cost.	$U[0, 1]$
q_j	Size of component j.	$U[10, 40]$
h_j	Size of data transferred between user and server j.	$U[1, 20]$
$g_{jj'}$	Size of data transferred between components j and j'	low :$U[1, 10]$ medium: $U[10, 100]$ high: $U[1000, 10000]$

We generate several problem instances with different values for n, the number of components of the application, and m, the number of servers in the network. The number of components for each application ranges from 2 to 100, while the number of servers ranges from 10 to 200. The reason for choosing these ranges is that in practice the number of components of an average application rarely exceeds 100 and most likely is on the lower part of the range considered here. Also, we assume that the number of time slots needed to run an application is 10. To generate the cost parameters defined in Section 3 we take into account the type of applications we consider.

Since the determinant factors in the performance of any algorithm for solving MCAPP are the server-component costs and the inter-component costs, we decided to generate the instances according to the value of a metric called *Inter-component cost to Server-component cost Ratio* (ISR). This metric is defined as the ratio of the average inter-component cost of each component (i.e., $\frac{\sum_{j=1}^{n}\sum_{j'=1}^{n}\bar{l}\cdot g_{jj'}\cdot r}{n}$) and the average server-component cost per assignment (i.e., $\frac{\sum_{i=1}^{m}\sum_{j=1}^{n}\omega_{ij}}{n\cdot m}$), where, $\bar{l}$ is the average distance between servers. Based on the value of ISR, we define three classes of applications with low, medium, and high ISR.

Table 5 shows the type of distributions used to generate the parameters characterizing the problem instances used in our simulation experiments. We consider different ranges for the distribution for three classes of applications. All the cost parameters for these three types of applications are the same, except for parameter $g_{jj'}$. This parameter indicates the inter-component communication intensiveness of the application. In Table 5, we denote by $U[x, y]$, the uniform distribution within interval $[x, y]$, and by $N(\mu, v)$, the normal distribution with mean μ and variance v. We assume that the cost of processing one unit of load on the servers is within the same range for all servers and does not vary significantly. Therefore, we use the normal distribution for the cost of processing. Similarly, we use the normal distribution for the processing requirement of the components.

We compare the performance of our algorithms, MATCH-MCAPP and G-MCAPP, with that of another algorithm called MATCH and with that of the optimal solution obtained by solving MCAPP-MINLP. The MATCH algorithm implements a variant of the Hungarian algorithm [37] and does not take into account the communication among com-

ponents when making the placement decisions. We compare with this algorithm in order to investigate the improvement in the quality of the solution due to considering the communication among components in the local search phase of MATCH-MCAPP.

For each type of instance, we determine the number of runs based on the observed variance of the results [39]. We observe that the standard deviation of the results for ten random instances is low. Thus, we execute MATCH-MCAPP, G-MCAPP, and MATCH algorithms for ten random instances (all the plots presented in the next section show the average values). The performance of the algorithms is evaluated by computing the *performance ratio*, PR, which is defined as the ratio of the value V^* of the optimal solution for MCAPP-MINLP, and V, the value of the solution obtained by a given algorithm, (i.e., $PR = \frac{V^*}{V}$). To obtain the optimal solution, we transform MCAPP-MINLP into an equivalent mixed integer linear program (called MCAPP-MILP) and solve it with the CPLEX solver. The transformation is performed by replacing $x_{ij} \cdot x_{i'j'}$ in the objective with a binary variable $y_{iji'j'}$, and adding the following constraints to the program,

$$x_{ij} + x_{i'j'} - 1 \leq y_{iji'j'} \quad \forall i, j, i', j' \tag{13}$$

$$y_{iji'j'} \in \{0, 1\} \quad \forall i, j, i', j' \tag{14}$$

These constraints guarantee that binary variable $y_{iji'j'}$ is 1, if both variables x_{ij} and $x_{i'j'}$ are 1; and 0 otherwise.

The MATCH-MCAPP, G-MCAPP, and MATCH algorithms are implemented in C++ and the experiments are conducted on an Intel 1.6GHz Core i5 with 8 GB RAM system. For solving MCAPP-MILP we use the CPLEX 12 solver provided by IBM ILOG CPLEX optimization studio for academics initiative [40].

6.2 Analysis of Results

In this section, we study the total placement cost of applications and compare the performance and the scalability of the algorithms for different types of applications with varying number of components and servers.

Performance with respect to the number of servers. We investigate the effect of the number of servers on the performance of MATCH-MCAPP and G-MCAPP algorithms considering the two mobility models. We characterize the performance of the algorithms using two main metrics, the performance ratio, and the execution time on a set of instances that runs the application over 10 time slots ($T = 10$) and consists of $n = 4$ components. We chose this type of instances in order to be able to solve them optimally using CPLEX and compare the performance of our algorithms with that of the optimal solution. We vary the number of servers from 10 to 100. These servers are chosen randomly from the data set. We select three types of instances for these experiments, instances with high inter-component costs, instances with medium inter-component costs, instances with low inter-component costs, and perform a detailed analysis of the results.

In Fig. 5, we plot the average execution time per time slot obtained by MATCH-MCAPP, G-MCAPP, and CPLEX under both mobility models and on those instances for different values of m using a logarithmic scale. In the plots, we denote by MATCH-MCAPP(TD) the cases in which MATCH-MCAPP is executed on instances generated using the trace-driven data sets, and by MATCH-MCAPP(RW), the cases in which MATCH-MCAPP is executed on the instances generated using the random-walk mobility model. We use a similar notation in the cases of G-MCAPP and CPLEX.

The execution time of CPLEX is several orders of magnitude higher than the execution times of both MATCH-MCAPP and G-MCAPP algorithms for all three types of instances. The execution time of MATCH-MCAPP and G-MCAPP, is under 1 millisecond for problem instances with small number of servers ($m < 40$), making them very suitable for deployment in real MEC systems.

We observe an increase of the execution time of MATCH-MCAPP with the increase in the number of servers. This is because of the cubic growth in terms of m of the running time of the algorithm. For example, in the case of instances with low inter-component communication, with $m = 20$ under the trace-driven model, the average execution time obtained by MATCH-MCAPP is around 0.08 milliseconds, while for $m = 80$, the average execution time is around 15 milliseconds. However, this execution time is reasonable because it is much less than the duration of a slot. Therefore, it will not make our algorithm a significant contributor to the overhead of placing the application components on edge servers.

Generally, the G-MCAPP algorithm obtains a lower execution time than MATCH-MCAPP. In all instances, the execution time obtained by G-MCAPP is under 1 millisecond. Also, we observe that in most cases, the total execution time of MATCH-MCAPP, G-MCAPP, and CPLEX under trace-driven data set is slightly greater than that under the random walk model. In fact, under the random walk model, a user changes his/her direction with the same probability in each time slot. Therefore, his/her distance from the servers may not change as significantly as the trace-driven case in which the user may only follow one direction for multiple consecutive time slots. Therefore, under the random walk model, the execution time of the algorithms is lower than that of trace-driven case, because the algorithms may not need to relocate components in each time slot.

In Fig. 6, we plot the performance ratio obtained by MATCH-MCAPP and G-MCAPP. In the case of low inter-component communication instances (Fig. 6a), the performance ratio obtained by MATCH-MCAPP is very close to 1, thus this algorithm obtains optimal solutions or solutions that are very close to the optimal. The reason is that the inter-component communication is low, and therefore, the solution obtained by the Hungarian algorithm (inside the MATCH-MCAPP) is very close to the optimal solution. For those instances, G-MCAPP has a lower performance ratio. However, the performance ratio obtained by this algorithm is higher than 0.87 in all cases. Furthermore, by increasing the number of servers, the quality of solutions obtained by G-MCAPP improves. The reason is that by increasing the number of servers, a higher percentage of servers are available for components at each iteration of G-MCAPP. Therefore, G-MCAPP explores more alternative placements and can make better decisions.

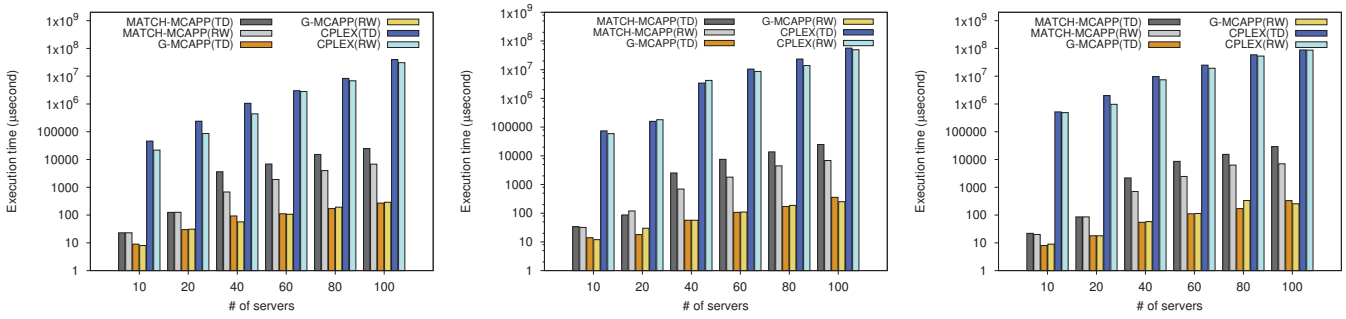

Fig. 5: Execution time (microseconds) vs. number of servers: (a) instances with low inter-component communication; (b) instances with medium inter-component communication; (c) instances with high inter-component communication.

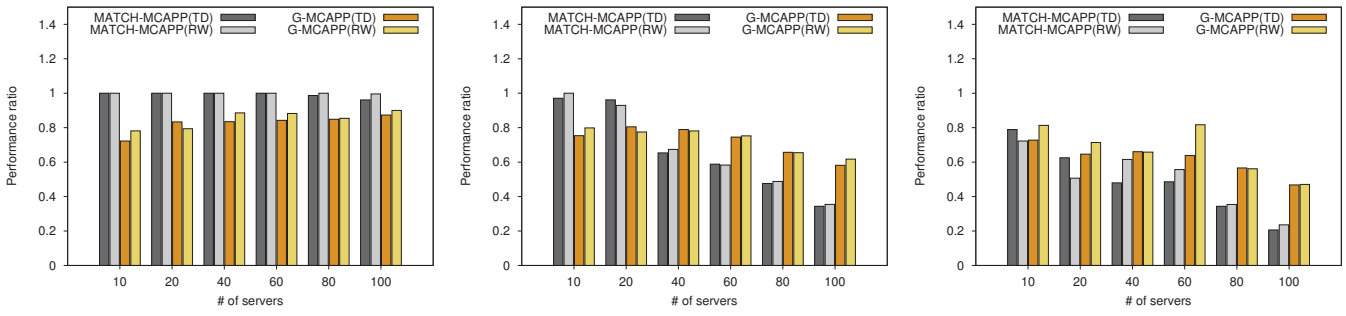

Fig. 6: Performance ratio vs. number of servers: (a) instances with low inter-component communication; (b) instances with medium inter-component communication; (c) instances with high inter-component communication.

As the amount of inter-component communication increases, the performance ratios of MATCH-MCAPP and G-MCAPP increase (Fig. 6b and 6c). We observe that G-MCAPP has a better performance compared to MATCH-MCAPP for higher inter-component communication cases. That means that G-MCAPP is able to obtain solutions that are closer to the optimal solution than those obtained by MATCH-MCAPP. As an example, for $m = 40$, for high inter-component communication case, the performance ratio of MATCH-MCAPP is 0.48, while that of G-MCAPP is 0.63. Also, we observe that the performance ratio decreases with the number of servers. Therefore, MATCH-MCAPP and G-MCAPP exhibit completely different behaviors compared to the low inter-component communication cases where the performance ratio increases by the increase of the number of servers. In fact, when the amount of inter-component communication is relatively high, each inefficient greedy decision can incur a significant amount of cost to the system. Thus, when the number of servers increases, there is a higher risk for local search/ greedy algorithms to make less efficient decisions.

Another important observation from Fig. 6b and 6c is that the performance ratio obtained by MATCH-MCAPP decreases faster than the performance ratio obtained by G-MCAPP. Also, from Fig. 6, we observe that the performance of MATCH-MCAPP and G-MCAPP under both trace-driven mobility data set and random walk model are consistent. In other words, the performance ratios obtained by the algorithms do not vary significantly under the considered mobility models. One reason for this consistency is the fact that our algorithms are relatively robust to the mobility behavior of users.

Performance with respect to the number of components. Next, we analyze the performance of MATCH-MCAPP and G-MCAPP by varying the number of components. We consider a set of instances that require running the application for 10 time slots ($T = 10$) and consist of 20 servers (we randomly choose 20 towers from the data set). We chose this type of instances in order to be able to solve them optimally using CPLEX and compare the performance of our algorithms with that of the optimal solution. We vary the number of components from 2 to 20. CPLEX was not able to solve some of the instances in feasible time, and thus, we were not able to determine the performance ratios for MATCH-MCAPP and G-MCAPP algorithms. In those cases we do not display the bars in the corresponding plots.

In Fig. 7, we plot the average execution time per time slot obtained by MATCH-MCAPP, G-MCAPP and CPLEX. The average execution time of the algorithms increases with the number of components. For example for instances with low inter-component communication and $n = 2$, the average execution time of MATCH-MCAPP is about 0.01 milliseconds under the trace-driven model, while for $n = 20$, it is about 0.1 milliseconds. Also, we observe that the execution time of G-MCAPP is much lower than the execution time of MATCH-MCAPP.

In Fig. 8, we plot the performance ratio obtained by MATCH-MCAPP and G-MCAPP algorithms under the both

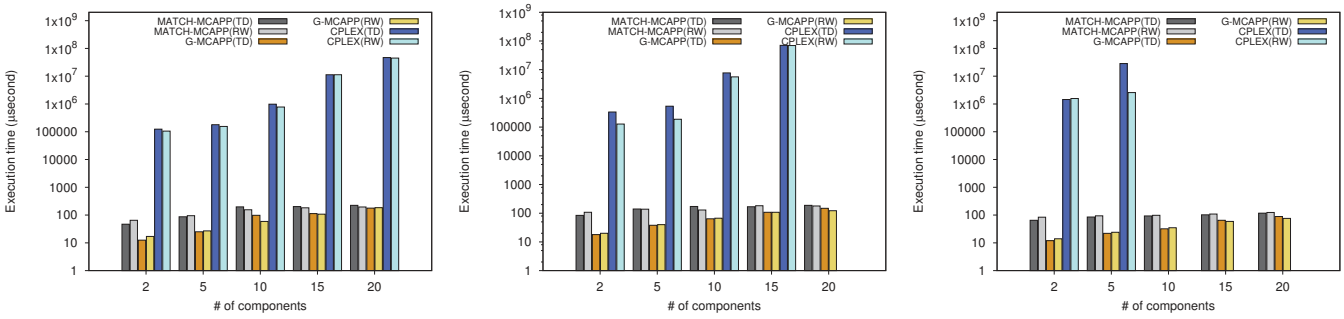

Fig. 7: Execution time (microseconds) vs. number of components: (a) instances with low inter-component communication; (b) instances with medium inter-component communication; (c) instances with high inter-component communication. (CPLEX was not able to determine the solutions for instances with 10, 15, and 20 components in feasible time, and thus, there are no bars in the plots for those cases)

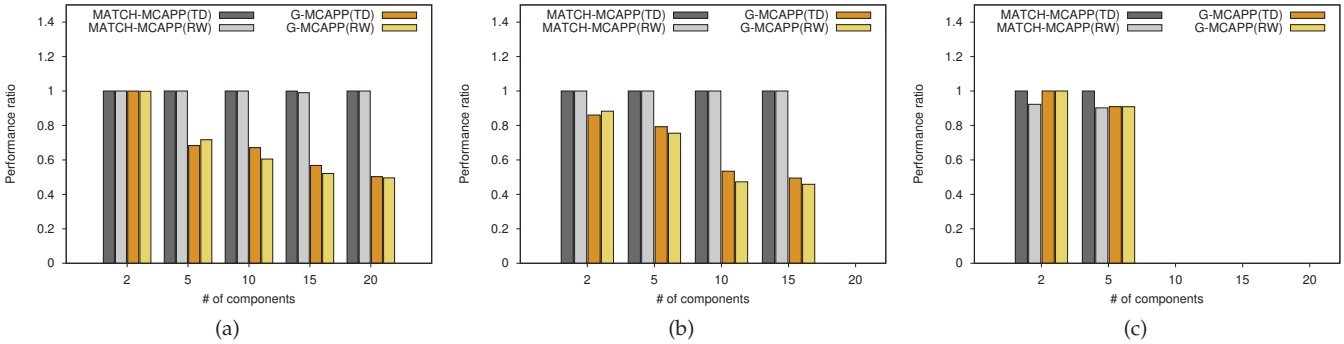

Fig. 8: Performance ratio vs. number of component: (a) instances with low inter-component communication; (b) instances with medium inter-component communication; (c) instances with high inter-component communication. (CPLEX was not able to determine the solutions for instances with 10, 15, and 20 components in feasible time, and thus, there are no bars in the plots for those cases)

mobility models. We observe that for instances with low or medium inter-component communication (Fig. 8a, Fig. 8b), MATCH-MCAPP outperforms G-MCAPP. Also, the performance ratio obtained by MATCH-MCAPP is very close to 1. For these instances, the performance ratio obtained by G-MCAPP decreases with the number of components. The reason is that, as the number of components increases, a smaller number of servers are available after each iteration of G-MCAPP. Therefore, the algorithm explores a smaller number of possible placements for each component.

Large-scale problem instances. We investigate the performance of the proposed algorithms for large-scale problem instances under the two mobility models. We consider large instances with a fixed number of components and servers ($n = 100$, $m = 200$) and fixed number of time slots ($T = 10$), and several values for ISR, ranging from 0.12 to 674. These instances with large number of components and servers are not expected to be encountered in practice, but we still consider them here to investigate the scalability of the algorithms. Since CPLEX is not feasible to use for solving such large instances, we will not compare the performance of our algorithms against the performance of the optimal solution obtained by CPLEX. Instead, we will compare the performance of our algorithm against that of MATCH (described in Section 6.1). In order to do this, we redefine the performance ratio as the ratio of the total cost obtained by MATCH and the total cost obtained by our proposed algorithms.

In Fig. 9a, we plot the average execution time per time slot obtained by MATCH-MCAPP and G-MCAPP. As expected, the average execution time per time slot of MATCH-MCAPP and G-MCAPP is not very sensitive to the value of ISR. The execution time of MATCH-MCAPP increases slowly with ISR. This is because more swaps may be performed in the local search when the inter-component communication increases. However, we observe that the execution time of G-MCAPP is not sensitive to the value of ISR. This is because the number of operations in the algorithm does not depend on the value of ISR. We also observe that, for large scale instances, the execution time of G-MCAPP is about two times greater than the execution time of MATCH-MCAPP; but it is still in a reasonable range compared to the duration of each time slot.

In Fig. 9b, we plot the performance ratio of MATCH-MCAPP and G-MCAPP under the two mobility models. We observe that both algorithms obtain better solutions compared to the MATCH algorithm. MATCH-MCAPP obtains solutions with a total cost around 45% lower than MATCH for most cases. For small values of ISR, since there is almost no communication among the components, MATCH-MCAPP behaves similarly to the Hungarian algorithm, that is, the local search step is not actually able to improve the solution beyond that obtained by matching. By increasing the value of ISR, the performance ratio of MATCH-MCAPP increases. This means that, the local search

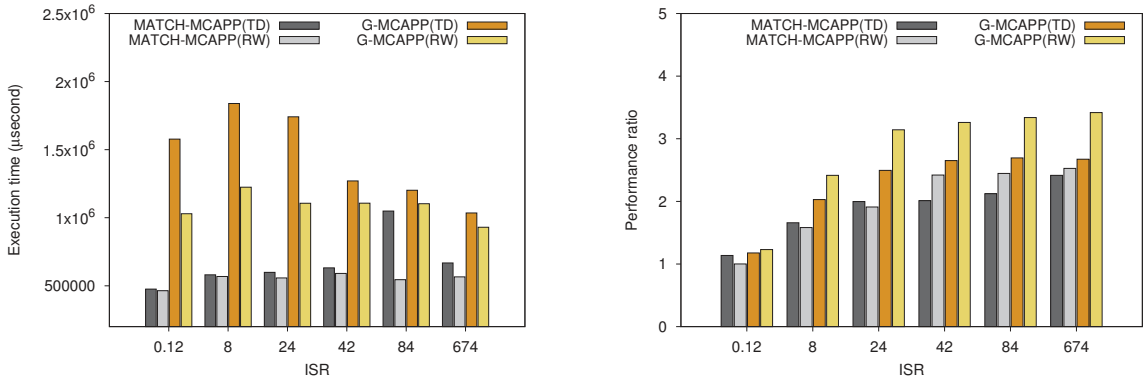

Fig. 9: The effect of ISR (large-scale instances): (a) Average execution time (microseconds) per time slot; (b) Performance ratio with respect to MATCH.

improves the solution obtained by the Hungarian algorithm. Furthermore, we observe that the performance of G-MCAPP is much better than MATCH-MCAPP. In most cases, the total cost obtained by G-MCAPP is around 60% less than total cost obtained by MATCH algorithm. This means that G-MCAPP obtains a better performance for large scale problem instances while it has a reasonable execution time.

According to the experimental results, MATCH-MCAPP and G-MCAPP obtain solutions that are very close to the optimal and require very low execution time per slot for reasonably large instances. For the average size instances, the ones we expect to encounter in practice, the proposed algorithms perform very well with respect to both the quality of the solutions and the execution time per time slot. Also, the performance of both algorithms is consistent under both the trace-driven mobility data set and the random walk model which indicates that the proposed algorithms are relatively robust to the mobility behavior of the users. MATCH-MCAPP is more suitable for MEC systems with a relatively low number of servers and components, and applications with less intensive communication among components. On the other hand, G-MCAPP is more suitable for MEC systems with a large number of servers and components, as well as for applications with intensive communication among components.

7 Conclusion

We addressed the problem of placement of multi-component applications in MEC systems. We formulated the problem as a Mixed Integer Non-Linear Program (MINLP) and developed two efficient algorithms for solving it. We performed extensive experiments to investigate the performance of the proposed algorithms. The results of these experiments indicated that the proposed algorithms obtain very good performance and require very low execution time, making them very suitable for deployment on MEC systems. For future work, we plan to design placement algorithms that take into account both the users' and providers' economic incentives when making placement decisions. A direct extension of this work is to consider settings in which a subset of the components of a single application are offloaded to a single server.

Acknowledgment

This paper is a revised and extended version of [41] presented at The Second ACM/IEEE Symposium on Edge Computing (SEC 2017). This research was supported in part by the NSF grant IIS-1724227.

References

[1] X. Fan, J. Cao, and H. Mao, "A survey of mobile cloud computing," *zTE Communications*, vol. 9, no. 1, pp. 4–8, 2011.
[2] A. Rudenko, P. Reiher, G. J. Popek, and G. H. Kuenning, "Saving portable computer battery power through remote process execution," *ACM SIGMOBILE Mobile Comp. and Comm. Rev.*, vol. 2, no. 1, pp. 19–26, 1998.
[3] J. Flinn and M. Satyanarayanan, "Energy-aware adaptation for mobile applications," in *Proc. 17th ACM Symp. on Operating Syst. Principles*, 1999, pp. 48–63.
[4] M. Satyanarayanan, "A brief history of cloud offload: A personal journey from odyssey through cyber foraging to cloudlets," *GetMobile: Mobile Comp. & Comm.*, vol. 18, no. 4, pp. 19–23, 2015.
[5] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The case for vm-based cloudlets in mobile computing," *IEEE Pervasive Computing*, vol. 8, no. 4, pp. 14–23, 2009.
[6] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proc. 1st Edition of the MCC Workshop on Mobile Cloud Comp.* ACM, 2012, pp. 13–16.
[7] T. Taleb and A. Ksentini, "Follow me cloud: interworking federated clouds and distributed mobile networks," *IEEE Network*, vol. 27, no. 5, pp. 12–19, 2013.
[8] M. Patel, B. Naughton, C. Chan, N. Sprecher, S. Abeta, A. Neal *et al.*, "Mobile-edge computing introductory technical white paper," *White Paper, Mobile-edge Computing (MEC) industry initiative*, 2014.
[9] Open Edge Computing, url = http://openedgecomputing.org.
[10] OpenFog Consortium, url = https://www.openfogconsortium.org.
[11] M. Piorkowski, N. Sarafijanovic-Djukic, and M. Grossglauser, "CRAWDAD dataset epfl/mobility (v. 2009-02-24)," Downloaded from https://crawdad.org/epfl/mobility/20090224, Feb. 2009.
[12] T. Camp, J. Boleng, and V. Davies, "A survey of mobility models for ad hoc network research," *Wireless Comm. & Mobile Comp.*, vol. 2, no. 5, pp. 483–502, 2002.
[13] D. Dutta, M. Kapralov, I. Post, and R. Shinde, "Embedding paths into trees: Vm placement to minimize congestion," in *European Symposium on Algorithms*. Springer, 2012, pp. 431–442.
[14] M. Chowdhury, M. R. Rahman, and R. Boutaba, "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Trans. Networking*, vol. 20, no. 1, pp. 206–219, 2012.
[15] S. Guo, B. Xiao, Y. Yang, and Y. Yang, "Energy-efficient dynamic offloading and resource scheduling in mobile cloud computing," in *Proc. 35th Annual IEEE Int. Conf. on Comp. Comm.*, 2016, pp. 1–9.
[16] D. Huang, P. Wang, and D. Niyato, "A dynamic offloading algorithm for mobile computing," *IEEE Trans. on Wireless Comm.*, vol. 11, no. 6, pp. 1991–1995, 2012.

[17] F. Berg, F. Dürr, and K. Rothermel, "Optimal predictive code offloading," in *Proc. 11th Int. Conf. on Mobile and Ubiquitous Syst.: Computing, Networking and Services*. ICST, 2014, pp. 1–10.
[18] J. Kwak, Y. Kim, J. Lee, and S. Chong, "Dream: Dynamic resource and task allocation for energy minimization in mobile cloud systems," *IEEE J. on Selected Areas in Comm.*, vol. 33, no. 12, pp. 2510–2523, Dec 2015.
[19] M. S. Elbamby, M. Bennis, and W. Saad, "Proactive edge computing in latency-constrained fog networks," *arXiv preprint arXiv:1704.06749*, 2017.
[20] T. Q. Dinh, J. Tang, Q. D. La, and T. Q. Quek, "Offloading in mobile edge computing: Task allocation and computational frequency scaling," *IEEE Trans. on Comm.*, vol. 65, no. 8, pp. 3571–3584, 2017.
[21] R. Kaewpuang, D. Niyato, P. Wang, and E. Hossain, "A framework for cooperative resource management in mobile cloud computing," *IEEE J. Selected Areas in Comm.*, vol. 31, no. 12, pp. 2685–2700, 2013.
[22] X. Lyu, H. Tian, C. Sengul, and P. Zhang, "Multiuser joint task offloading and resource optimization in proximate clouds," *IEEE Trans. Vehicular Technology*, vol. 66, no. 4, pp. 3435–3447, 2017.
[23] A. Ksentini, T. Taleb, and M. Chen, "A markov decision process-based service migration procedure for follow me cloud," in *Proc. IEEE Int. Conf. on Comm.*, 2014, pp. 1350–1354.
[24] R. Urgaonkar, S. Wang, T. He, M. Zafer, K. Chan, and K. K. Leung, "Dynamic service migration and workload scheduling in edge-clouds," *Performance Evaluation*, vol. 91, pp. 205 – 228, 2015.
[25] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *Proc. IFIP Networking Conf.* IEEE, 2015, pp. 1–9.
[26] S. Wang, M. Zafer, and K. K. Leung, "Online placement of multi-component applications in edge computing environments," *IEEE Access*, vol. 5, pp. 2514–2533, 2017.
[27] J. Pei, P. Hong, K. Xue, and D. Li, "Efficiently embedding service function chains with dynamic virtual network function placement in geo-distributed cloud system," *IEEE Trans. Parallel and Distributed Systems*, vol. 30, no. 10, pp. 2179–2192, 2018.
[28] Z. Zhou, Q. Wu, and X. Chen, "Online orchestration of cross-edge service function chaining for cost-efficient edge computing," *IEEE J. Selected Areas in Comm.*, vol. 37, no. 8, pp. 1866–1880, 2019.
[29] M. Jia, J. Cao, and W. Liang, "Optimal cloudlet placement and user to cloudlet allocation in wireless metropolitan area networks," *IEEE Trans. Cloud Computing*, vol. 5, no. 4, pp. 725–737, 2017.
[30] Z. Xu, W. Liang, W. Xu, M. Jia, and S. Guo, "Efficient algorithms for capacitated cloudlet placements," *IEEE Trans. Parallel and Distributed Systems*, vol. 27, no. 10, pp. 2866–2880, 2016.
[31] A. Ceselli, M. Premoli, and S. Secci, "Mobile edge cloud network design optimization," *IEEE/ACM Trans. Networking*, vol. 25, no. 3, pp. 1818–1831, 2017.
[32] S. Wang, Y. Zhao, J. Xu, J. Yuan, and C.-H. Hsu, "Edge server placement in mobile edge computing," *J. Parallel and Distributed Computing*, vol. 127, pp. 160 – 168, 2019.
[33] H. Zhao, S. Deng, Z. Liu, J. Yin, and S. Dustdar, "Distributed redundancy scheduling for microservice-based applications at the edge," *IEEE Trans. on Services Computing*, 2020.
[34] S. Deng, Z. Xiang, P. Zhao, J. Taheri, H. Gao, J. Yin, and A. Y. Zomaya, "Dynamical resource allocation in edge for trustable internet-of-things systems: A reinforcement learning method," *IEEE Trans. on Industrial Informatics*, vol. 16, no. 9, pp. 6103–6113, 2020.
[35] S. Deng, Z. Xiang, J. Taheri, K. A. Mohammad, J. Yin, A. Zomaya, and S. Dustdar, "Optimal application deployment in resource constrained distributed edges," *IEEE Trans. on Mobile Computing*, 2020.
[36] Z. Xiang, S. Deng, J. Taheri, and A. Zomaya, "Dynamical service deployment and replacement in resource-constrained edges," *Mobile Networks and Applications*, vol. 25, no. 2, pp. 674–689, 2020.
[37] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Res. Logistics Quarterly*, vol. 2, no. 1-2, pp. 83–97, 1955.
[38] (2017) GPS visualizer. [Online]. Available: http://www.gpsvisualizer.com/
[39] D. S. Johnson, "A theoretican's guide to the experimental analysis of algorithms," *Data structures, near neighbor searches, and methodology: fifth ann sixth DIMACS implementation challenges*, vol. 59, pp. 215–250, 2002.
[40] (2009) IBM ILOG CPLEX V12.1 user's manual. [Online]. Available: ftp://ftp.software.ibm.com/software/websphere/ilog/docs/
[41] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proc. 2nd ACM/IEEE Symp. on Edge Computing*, 2017, pp. 5:1–5:11.

BIOGRAPHIES

Tayebeh Bahreini is currently a Ph.D. candidate at Wayne State University, Department of Computer Science. She graduated from Shahed University in Iran with a M.Sc. degree in Computer Engineering, in 2014. She received her B.Sc. degree in Computer Science from University of Isfahan, Iran, in 2010. Her main research interests are distributed systems, approximation algorithms, parallel computing, and game theory. She is the recipient of the 2019 National Center for Women & Information Technology (NCWIT) Collegiate National Award. She was selected as a *2019 Top Ten Women in Edge* by the Edge Computing World organization for her contribution to research in edge computing. She is a student member of the ACM and the IEEE.

Daniel Grosu received the Diploma in engineering (automatic control and industrial informatics) from the Technical University of Iaşi, Romania, in 1994 and the MSc and PhD degrees in computer science from the University of Texas at San Antonio in 2002 and 2003, respectively. Currently, he is an associate professor in the Department of Computer Science, Wayne State University, Detroit. His research interests include parallel and distributed computing, approximation algorithms, and topics at the border of computer science, game theory and economics. He has published more than one hundred peer-reviewed papers in the above areas. He has served on the program and steering committees of several international meetings in parallel and distributed computing such as ICDCS, CLOUD, ICPP and NetEcon. He is a senior member of the ACM, the IEEE, and the IEEE Computer Society.