Online Placement of Multi-Component Applications in Edge Computing Environments

Shiqiang Wang, *Member, IEEE,* Murtaza Zafer, *Member, IEEE,* and Kin K. Leung, *Fellow, IEEE*

***Abstract*—Mobile edge computing is a new cloud computing paradigm which makes use of small-sized edge-clouds to provide real-time services to users. These mobile edge-clouds (MECs) are located in close proximity to users, thus enabling users to seamlessly access applications running on MECs. Due to the co-existence of the core (centralized) cloud, users, and one or multiple layers of MECs, an important problem is to decide where (on which computational entity) to place different components of an application. This problem, known as the application or workload placement problem, is notoriously hard, and therefore, heuristic algorithms without performance guarantees are generally employed in common practice, which may unknowingly suffer from poor performance as compared to the optimal solution. In this paper, we address the application placement problem and focus on developing algorithms with provable performance bounds. We model the user application as an application graph and the physical computing system as a physical graph, with resource demands/availabilities annotated on these graphs. We first consider the placement of a linear application graph and propose an algorithm for finding its optimal solution. Using this result, we then generalize the formulation and obtain online approximation algorithms with polynomial-logarithmic (poly-log) competitive ratio for tree application graph placement. We jointly consider node and link assignment, and incorporate multiple types of computational resources at nodes.**

***Index Terms*—Cloud computing, graph mapping, mobile edge-cloud (MEC), online approximation algorithm, optimization theory**

I. Introduction

Mobile applications relying on cloud computing became increasingly popular in the recent years [2], [3]. Different from traditional standalone applications that run solely on a mobile device, a cloud-based application has one or multiple components running in the cloud, which are connected to another component running on the handheld device and they jointly constitute an application accessible to the mobile user.

This research was sponsored in part by the U.S. Army Research Laboratory and the U.K. Ministry of Defence and was accomplished under Agreement Number W911NF-06-3-0001 and W911NF-16-3-0001. The views and conclusions contained in this document are those of the author(s) and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon.

S. Wang is with IBM T. J. Watson Research Center, Yorktown Heights, NY, United States. Email: wangshiq@us.ibm.com

M. Zafer is with Nyansa Inc., Palo Alto, CA, United States. E-mail: murtaza.zafer.us@ieee.org

K. K. Leung is with the Department of Electrical and Electronic Engineering, Imperial College London, United Kingdom. E-mail: kin.leung@imperial.ac.uk

Part of the material presented in this paper appeared in S. Wang's Ph.D. thesis [1].

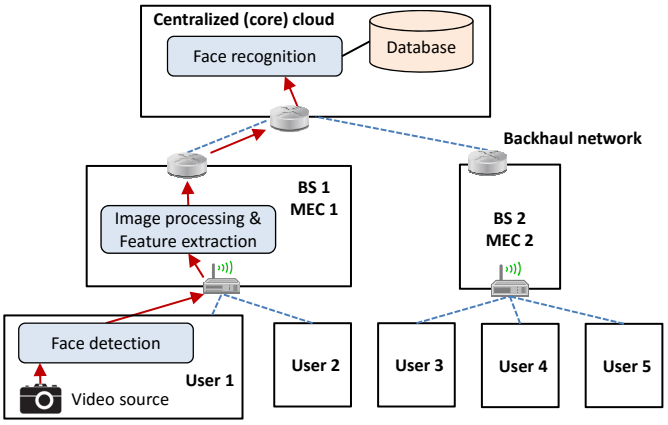

Figure 1. Application scenario with mobile edge-clouds (MECs). Example scenario with face recognition application, where the dashed lines stand for physical communication links and red arrows stand for the data transmission path.

Examples of cloud-based mobile applications include map, storage, and video streaming services [4], [5]. They all require high data processing/storage capability that cannot be satisfied on handheld devices alone, thus it is necessary to run part of the application in the cloud.

Traditionally, clouds are located in centralized data-centers. One problem with cloud-based applications is therefore the long-distance communication between the user device and the cloud, which may cause intermittent connectivity and long latency that cannot satisfy the requirements of emerging interactive applications such as real-time face recognition and online gaming [6]. To tackle this issue, *mobile edge-cloud (MEC)* has been proposed recently [7], [8]. The idea is to have small cloud-like entities (i.e., MECs) deployed at the edge of communication networks, which can run part or all of the application components. These MECs are located close to user locations, enabling users to have seamless and low-latency access to cloud services. For example, they can co-locate with edge devices such as Wi-Fi access points or cellular base stations (BSs), as shown in Fig. 1, forming up a hierarchy together with the centralized cloud and mobile users. The concept of MEC is similar to cloudlet [9], fog computing [10], [11], follow me cloud [12], and small cell cloud [13].

Although MECs are promising, there are limitations. In particular, they have a significantly lower processing and storage capability compared to the core (centralized) cloud, thus it is usually infeasible to completely abandon the core cloud and run everything on MECs. An important problem is therefore to decide where (i.e., whether on the core cloud,

MEC, or mobile device) to place different processing and storage components of an application. This is referred to as the *application placement problem*, which is a non-trivial problem as illustrated by the example below.

A. Motivating Example

Consider an application which recognizes faces from a real-time video stream captured by the camera of a hand-held device. As shown in Fig. 1, we can decompose this application into one storage component (the database) and three different processing components including face detection (FD), image processing and feature extraction (IPFE), and face recognition (FR). The FD component finds areas of an image (a frame of the video stream) that contains faces. This part of image is sent to IPFE for further processing. The main job of IPFE is to filter out noise in the image and extract useful features for recognizing the person from its face. These features are sent to FR for matching with a large set of known features of different persons' faces stored in the database.

Fig. 1 shows one possible placement of FD, IPFE, FR, and the database onto the hierarchical cloud architecture. This can be a good placement in some cases, but may not be a good placement in other cases.

For example, the benefit of running FD on the mobile device instead of MEC is that it reduces the amount of data that need to be transferred between the mobile device and MEC. However, in cases where the mobile device's processing capability is strictly limited but there is a reasonably high bandwidth connection between the mobile device and MEC, it is can be good to place FD on the MEC. Having the database in the core cloud can be beneficial because it can contain a large amount of data infeasible for the MEC to store. In this case, FR should also be in the core cloud because it needs to frequently query the database. However, if the database is relatively small and has locally generated contents, we may want to place the database and FR onto the MEC instead of the core cloud, as this reduces the backhaul network load.

We see that even with this simple application, it is non-straightforward to conceptually find the best placement, while many realistic applications such as streaming, multicasting, and data aggregation [14]–[16] are much more complex. We also note that MECs can be attached to devices at different cellular network layers [8], yielding a hierarchical cloud structure with more than three layers. Meanwhile, there usually exist multiple applications that are instantiated at the cloud system over time. All these aspects motivate us to consider the application placement problem in a rigorous optimization framework where applications arrive in an *online* manner, i.e., they arrive sequentially over time and we do not have knowledge on the characteristics of future applications at any point of time.

We will abstract the application placement problem as the problem of placing application graphs, which represent application components and the communication among these components, onto a physical graph, which represents the computing devices and communication links in the physical system, as shown in Fig. 2. For example, the face recognition application above can be abstracted as an application graph with four nodes connected in a chain (line), where each of the nodes represents the database, FR, IPFE, and FD in sequential order. The detailed problem formulation will be presented in Section II.

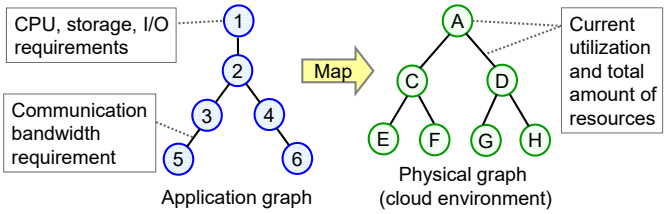

Figure 2. The application placement problem.

B. Related Work

A body of existing work on application placement and scheduling in MECs has considered applications with two components, one running on a cloud (can be either MEC or core cloud) and the other running on the user [12], [17]–[19]. Another body of existing work, which is also known under the term "cloud offloading", usually involves only two physical computing entities (i.e., the mobile device and the cloud) [20]–[22]. Multi-component applications that can be deployed accross one or multiple levels of MECs and core cloud(s) have not been considered, whereas such applications widely exist in practice because MEC servers can locate at multiple network equipments in different hierarchical levels [8].

The multi-component application placement problem has been studied mainly in data-center settings. Because this problem is NP-hard even for simple graphs (as we discuss later), a common practice is to employ heuristic algorithms without performance guarantees [23], [24], which may unknowingly suffer from poor performance as compared to the optimal solution. Only a very limited amount of existing work followed a rigorous theoretical framework from approximation algorithms [25] and competitive analysis [26], and proposed approximation algorithms (i.e., approximately optimal algorithms) with provable approximation/competitive ratios[1] for the application placement problem, in particular when it involves both node and link placements.

In [27], the authors proposed an algorithm for minimizing the sum cost while considering load balancing, which has an approximate approximation ratio of $O(N)$, where N is the number of nodes in the physical graph. The algorithm is based on linear program (LP) relaxation, and only allows one node in each application graph to be placed on a particular physical node; thus, excluding server resource sharing among different nodes in one application graph. It is shown that the approximation ratio of this algorithm is $O(N)$, which is trivial because one would achieve the same approximation ratio when

[1]For a minimization problem, the *competitive ratio* is defined as an upper bound of the online approximation algorithm's cost to the true optimal cost that can be obtained from an offline placement, where the offline placement considers all application graphs simultaneously instead of considering them arriving over time. The definition of approximation ratio is the same but it is for offline problems.

placing the whole application graph onto a single physical node instead of distributing it across the whole physical graph.

A theoretical work in [28] proposed an algorithm with $N^{O(D)}$ time-complexity and an approximation ratio of $\delta = O(D^2 \log(ND))$ for placing a tree application graph with D levels of nodes onto a physical graph. It uses LP relaxation and its goal is to minimize the sum cost. Based on this algorithm, the authors presented an online algorithm for minimizing the maximum load on each node and link, which is $O(\delta \log(N))$-competitive when the application lifetimes are equal. The LP formulation in [28] is complex and requires $N^{O(D)}$ variables and constraints. This means when D is not a constant, the space-complexity (specifying the required memory size of the algorithm) is exponential in D.

Another related theoretical work which proposed an LP-based method for offline placement of paths into trees in data-center networks was reported in [29]. Here, the application nodes can only be placed onto the leaves of a tree physical graph, and the goal is to minimize link congestion. In our problem, the application nodes are distributed across users, MECs, and core cloud, thus they should not be only placed at the leaves of a tree so the problem formulation in [29] is inapplicable to our scenario. Additionally, [29] only focuses on minimizing link congestion. The load balancing of nodes is not considered as part of the objective; only the capacity limits of nodes are considered.

Some other related work focuses on graph partitioning, such as [30] and [31], where the physical graph is defined as a complete graph with edge costs associated with the distance or latency between physical servers. Such an abstraction combines multiple network links into one (abstract) physical edge, which may hide the actual status of individual links along a path.

A related problem that has emerged recently is the service chain embedding problem [32]–[34]. Motivated by network function virtualization (NFV) applications, the goal is to place a linear application graph between fixed source and destination physical nodes, so that a series of operations are performed on data packets sent from the source to the destination. Within this body of work, only [34] has studied the competitive ratio of online placement, which, however, does not consider link placement optimization.

One important aspect to note is that most existing work, including [27], [29]–[33], do not specifically consider the online operation of the algorithms. Although some of them implicitly claim that one can apply the algorithm repeatedly for each newly arrived application, the competitive ratio of such procedure is unclear. To the best of our knowledge, [28] is the only work that studied the competitive ratio of the online application placement problem that considers *both node and link placements*.

C. Our Approach

In this paper, we focus on the MEC context and propose algorithms for solving the online application placement problem with provable competitive ratios. Different from [28], our approach is *not* based on LP relaxation. Instead, our algorithms are built upon a baseline algorithm that provides an *optimal* solution to the placement of a linear application graph (i.e., an application graph that is a line). This is an important novelty in contrast to [28] where no optimal solution was presented for any scenario. Many applications expected to run in an MEC environment can be abstracted as hierarchical graphs, and the simplest case of such a hierarchical graph is a line, such as the face recognition example in Section I-A. Therefore, the placement of a linear application graph is an important problem in the context of MECs.

Another novelty in our work, compared to [28] and most other approaches based on LP relaxation, is that our solution approach is *decomposable* into multiple small building blocks. This makes it easy to extend our proposed algorithms to a distributed solution in the future, which would be very beneficial for reducing the amount of necessary control information exchange among different cloud entities in a distributed cloud environment containing MECs. This decomposable feature also makes it easier to use these algorithms as a sub-procedure for solving a larger problem.

It is also worth noting that the analytical methodology we use in this paper is new compared to existing techniques such as LP relaxation, thus we enhance the set of tools for online optimization. The theoretical analysis in this paper also provides insights on the features and difficulty of the problem, which can guide future practical implementations. In addition, the proposed algorithms themselves are relatively easy to implement in practice.

D. Main Results

We propose non-LP based approximation algorithms for online application placement in this paper. The general problem of application placement is hard to approximate [28], [29], [32], [35]. For example, [32] has shown that theoretically, there exists no polynomial-time approximation algorithm with bounded approximation ratio for a service chain embedding problem that considers linear application graph and general physical graphs, and has both node and link capacity constraints. Therefore, similar to related work [27]–[34], we make a few simplifications to make the problem tractable. These simplifications are driven by realistic MEC settings and their motivations are described as follows.

Throughout this paper, we focus on application and physical graphs that have tree topologies. This is due to the consideration that a tree application graph models a wide range of MEC applications that involve a hierarchical set of processes (or virtual machines), including streaming, multicasting, and data aggregation applications [14]–[16] such as the exemplar face recognition application presented earlier. For the physical system, we consider tree physical graphs due to the hierarchical nature of MEC environment (see Fig. 1). We note that the algorithms we propose in this paper *also works with several classes of non-tree graphs* and an example will be given in Section VI. For ease of presentation, we mainly focus on tree graphs in this paper.

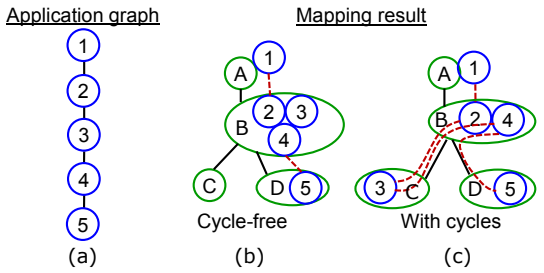

Figure 3. Mapping with and without cycles. In this example, the path in the application graph is between application node 1 and application node 5.

In the tree application graph, if we consider any path from the root to a leaf, we only allow those assignments[2] where the application nodes along this path are assigned in their respective order on a sub-path of the physical topology (multiple application nodes may still be placed onto one physical node), thus, creating a "cycle-free" placement. Figure 3 illustrates this placement. Let nodes 1 to 5 denote the application nodes along a path in the application-graph topology. The cycle-free placement of this path onto a sub-path of the physical network ensures the order is preserved (as shown in Fig. 3(b)), whereas the order is not preserved in Fig. 3(c). A cycle-free placement has a clear motivation of avoiding cyclic communication among the application nodes. For example, for the placement in Fig. 3(c), application nodes 2 and 4 are placed on physical node B, while application node 3 is placed on physical node C. In this case, the physical link B–C carries the data of application links 2–3 and 3–4 in a circular fashion. Such traffic can be naturally avoided with a cycle-free mapping (Fig. 3(b)), thus relieving congestion on the communication links. As we will see in the simulations in Section V, the cycle-free constraint still allows the proposed scheme to outperform some other comparable schemes that allow cycles. Further discussion on the approximation ratio associated with cycle-free restriction is given in Appendix A.

In this paper, for the purpose of describing the algorithms, we classify an application node as a *junction node* in the tree application graph when it has two or more children. These junction nodes may represent data splitting or joining processes for multiple data streams. In some cases, they may have pre-specified placement, because they serve multiple data streams that may be associated with different end-users, and individual data streams may arrive dynamically in an online fashion. Our work first considers cases where the placements of all junction nodes (if any) are pre-specified, and then extends the results to the general case where some junction nodes are not placed beforehand. A linear application graph (such as the exemplar face recognition application in Section I-A) has no junction nodes and it falls into the first category.

For the aforementioned scenario, we obtain the following main results for the problem of application placement with the goal of load balancing among physical nodes and edges:

1) An optimal offline algorithm for placing a single application graph which is a linear graph, with $O(V^3N^2)$ time-complexity and $O(VN(V+N))$ space-complexity, where the application graph has V nodes and the physical graph has N nodes.
2) An online approximation algorithm for placing single or multiple tree application graphs, in which the placements of all junction nodes are pre-specified, i.e., their placements are given. This algorithm has a time-complexity of $O(V^3N^2)$ and a space-complexity of $O(VN(V+N))$ for each application graph placement; its competitive ratio is $O(\log N)$.
3) An online approximation algorithm for placing single or multiple tree application graphs, in which the placements of some junction nodes are *not* pre-specified. This algorithm has a time-complexity of $O(V^3N^{2+H})$ and a space-complexity of $O(VN^{1+H}(V+N))$ for each application graph placement; its competitive ratio is $O(\log^{1+H} N)$, where H is the maximum number of junction nodes without given placement on any single path from the root to a leaf in the application graph. Note that we always have $H \leq D$, where D is the depth of the tree application graph.

Our work considers multiple types of resources on each physical node, such as CPU, storage, and I/O resources. The proposed algorithms can work with domain constraints which restrict the set of physical nodes that a particular application node can be assigned to. The exact algorithm for single line placement can also incorporate conflict constraints where some assignments are not allowed for a pair of adjacent application nodes that are connected by an application edge; such constraints may arise in practice due to security policies as discussed in [23].

The remainder of this paper is organized in the following: Section II describes the problem formulation. Section III presents the exact optimal placement algorithm for single linear application graph. Online approximation algorithms for tree-to-tree placement are discussed in Section IV. Section V shows numerical results. Some insights and observations are discussed in Section VI. Section VII draws conclusions.

II. Problem Formulation

A. Definitions

We consider the placement of application graphs onto a physical graph, where the application graphs represent applications that may arrive in an online manner. In the following, we introduce some notations that will be used in this paper.

Application Graph: An application is abstracted as a graph, in which nodes represent the processing/computational modules of the application (such as virtual machines or containers containing running application processes), and edges represent the communication demand between nodes (such as information sharing among different application processes). Each node $v \in \mathcal{V}$ in the application graph $\mathcal{R} = (\mathcal{V}, \mathcal{E})$ is associated with parameters that represent the computational resource (of K different types) demands of node v. Similarly, each edge $e \in \mathcal{E}$ is associated with a communication bandwidth demand. The notation $e = (v_1, v_2)$ denotes that application edge e connects application nodes v_1 and v_2. The application graph

[2]We interchangeably use the terms "placement", "assignment", and "mapping" in this paper.

$\mathcal{R}$ can be either a directed or an undirected graph. If it is a directed graph, the direction of edges specify the direction of data communication; if it is an undirected graph, data communication can occur in either direction along application edges.

Physical Graph: The physical computing system is also abstracted as a graph, with nodes denoting computing devices[3] and edges denoting communication links between nodes. Each node $n \in \mathcal{N}$ in the physical graph $\mathcal{Y} = (\mathcal{N}, \mathcal{L})$ has K different types of computational resources, and each edge $l \in \mathcal{L}$ has communication resource. A physical node can also represent a network device such as a router or switch with zero computational resource. We use the notation $l = (n_1, n_2)$ to denote that physical link l connects physical nodes n_1 and n_2. Similar to the application graph, the physical graph can be either directed or undirected, depending on whether the physical links are bidirectional (i.e., communication in both directions share the same link) or single-directional (i.e., communication in each direction has a separate link).

Because we consider multiple application graphs in this paper, we denote the tree application graph for the ith application arrival as $\mathcal{R}(i) = (\mathcal{V}(i), \mathcal{E}(i))$. Throughout this paper, we define $V = |\mathcal{V}|$, $E = |\mathcal{E}|$, $N = |\mathcal{N}|$, and $L = |\mathcal{L}|$, where $|\cdot|$ denotes the number of elements in the corresponding set.

We consider undirected application and physical graphs in the problem formulation, which means that data can flow in any direction on an edge, but the proposed algorithms can be easily extended to many types of directed graphs. For example, when the tree application graph is directed and the tree physical graph is undirected, we can merge the two application edges that share the same end nodes in different directions into one edge, and focus on the merged undirected application graph for the purpose of finding optimal placement. This does not affect the optimality because for any placement of application nodes, there is a unique path connecting two different application nodes due to the cycle-free constraint and the tree structure of physical graphs. Thus, application edges in both directions connecting the same pair of application nodes have to be placed along the same path on the physical graph.

Costs: For the ith application, the weighted cost (where the weighting factor can serve as a normalization to the total resource capacity) for type $k \in \{1, 2, ..., K\}$ resource of placing v to n is denoted by $d_{v\to n,k}(i)$. Similarly, the weighted communication bandwidth cost of assigning e to l is denoted by $b_{e\to l}(i)$. The edge cost is also defined for a dummy link $l = (n, n)$, namely a non-existing link that connects the same node, to take into account the additional cost when placing two application nodes on one physical node. It is also worth noting that an application edge may be placed onto multiple physical links that form a path.

Remark: The cost of placing the same application node (or edge) onto different physical nodes (or edges) can be different. This is partly because different physical nodes and edges may have different resource capacities, and therefore different weighting factors for cost computation. It can also be due to the domain and conflict constraints as mentioned earlier. If some mapping is not allowed, then we can set the corresponding mapping cost to infinity. Hence, our cost definitions allow us to model a wide range of access-control/security policies.

[3]Multiple individual servers can be seen as a single entity if they constitute a single cloud.

Mapping: A mapping is specified by $\pi : \mathcal{V} \to \mathcal{N}$. Because we consider tree physical graphs with the cycle-free restriction, there exists only one path between two nodes in the physical graph, and we use (n_1, n_2) to denote either the link or path between nodes n_1 and n_2. We use the notation $l \in (n_1, n_2)$ to denote that link l is included in path (n_1, n_2). The placement of nodes automatically determines the placement of edges.

In a successive placement of the 1st up to the ith application, each physical node $n \in \mathcal{N}$ has an aggregated weighted cost of

$$p_{n,k}(i) = \sum_{j=1}^{i} \sum_{v:\pi(v)=n} d_{v\to n,k}(j), \tag{1}$$

where the second sum is over all v that are mapped to n. Equation (1) gives the total cost of type k resource requested by all application nodes that are placed on node n, upto the ith application. Similarly, each physical edge $l \in \mathcal{L}$ has an aggregated weighted cost of

$$q_l(i) = \sum_{j=1}^{i} \sum_{e=(v_1,v_2):(\pi(v_1),\pi(v_2))\ni l} b_{e\to l}(j), \tag{2}$$

where the second sum is over all application edges $e = (v_1, v_2)$ for which the path between the physical nodes $\pi(v_1)$ and $\pi(v_2)$ (which v_1 and v_2 are respectively mapped to) includes the link l.

B. Objective Function

The optimization objective in this paper is load balancing for which the objective function is defined as

$$\min_{\pi} \max \left\{ \max_{k,n} p_{n,k}(M); \max_{l} q_l(M) \right\}, \tag{3}$$

where M is the total number of applications (application graphs). Equation (3) aims to minimize the maximum weighted cost on each physical node and link, ensuring that no single element gets overloaded and becomes a point of failure, which is important especially in the presence of bursty traffic. Such an objective is widely used in the literature [36], [37].

Remark: While we choose the objective function (3) in this paper, we do realize that there can be other objectives as well, such as minimizing the total resource consumption. We note that the exact algorithm for the placement of a single linear application graph *can be generalized to a wide class of other objective functions* as will be discussed in Section III-E. For simplicity, we restrict our attention to the objective function in (3) in most parts of our discussion. We also note that our objective function incorporates both node and link resource consumptions, which is important in MEC environments where both node (server) and communication link conditions are closely related to the service performance.

A Note on Capacity Limit: For simplicity, we do not impose capacity constraints on physical nodes and links in

the optimization problem defined in (3), because even without the capacity constraint, the problem is very hard as we will see later in this paper. However, because the resource demand of each application node and link is specified in every application graph, the total resource consumption at a particular physical node/link can be calculated by summing up the resource demands of application nodes/links that are placed on it. Therefore, an algorithm can easily check within polynomial time whether the current placement violates the capacity limits. If such a violation occurs, it can simply reject the newly arrived application graph.

In most practical cases, the costs of node and link placements should be defined as proportional to the resource occupation when performing such placement, with weights inversely proportional to the capacity of the particular type of resource. With such a definition, the objective function (3) essentially tries to place as many application graphs as possible without increasing the maximum resource occupation (normalized by the resource capacity) among all physical nodes and links. Thus, the placement result should utilize the available resource reasonably well. A more rigorous analysis on the impact of capacity limit is left as future work.

III. Basic Assignment Unit: Single Linear Application Graph Placement

We first consider the assignment of a single linear application graph (i.e., the application nodes are connected in a line), where the goal is to find the best placement of application nodes onto a path in the tree physical graph under the cycle-free constraint (see Fig. 3). The solution to this problem forms the building block of other more sophisticated algorithms presented later. As discussed next, we develop an algorithm that can find the optimal solution to this problem. We omit the application index i in this section because we focus on a single application, i.e., $M = 1$, here.

A. Sub-Problem Formulation

Without loss of generality, we assume that $\mathcal{V}$ and $\mathcal{N}$ are indexed sets, and we use v to exchangeably denote elements and indices of application nodes in $\mathcal{V}$, and use n to exchangeably denote elements and indices of physical nodes in $\mathcal{N}$. This index (starting from 1 for the root node) is determined by the topology of the graph. In particular, it can be determined via a breadth-first or depth-first indexing on the tree graph (note that linear graphs are a special type of tree graphs). From this it follows that, if n_1 is a parent of n_2, then we must have $n_1 < n_2$. The same holds for the application nodes $\mathcal{V}$.

With this setting, the edge cost can be combined together with the cycle-free constraint into a single definition of pairwise costs. The weighted pairwise cost of placing $v-1$ to n_1 and v to n_2 is denoted by $c_{(v-1,v)\to(n_1,n_2)}$, and it takes the following values with $v \geq 2$:

- If the path from n_1 to n_2 traverses some $n < n_1$, in which case the cycle-free assumption is violated, then $c_{(v-1,v)\to(n_1,n_2)} = \infty$.
- Otherwise,

$$c_{(v-1,v)\to(n_1,n_2)} = \max_{l\in(n_1,n_2)} b_{(v-1,v)\to l}\Big|_{(\pi(v-1),\pi(v))\ni l}. \tag{4}$$

The maximum operator in (4) follows from the fact that, in the single line placement, at most one application edge can be placed onto a physical link. Also recall that the edge cost definition incorporates dummy links such as $l = (n, n)$, thus there always exists $l \in (n_1, n_2)$ even if $n_1 = n_2$.

Then, the optimization problem (3) with $M = 1$ becomes

$$\min_{\pi} \max \left\{ \max_{k,n} \sum_{v:\pi(v)=n} d_{v\to n,k}; \max_{(v-1,v)\in\mathcal{E}} c_{(v-1,v)\to(\pi(v-1),\pi(v))} \right\}. \tag{5}$$

The last maximum operator in (5) takes the maximum among all application edges (rather than physical links), because when combined with the maximum in (4), it essentially computes the maximum among all physical links that are used for data transmission under the mapping π.

B. Decomposing the Objective Function

In this subsection, we decompose the objective function in (5) to obtain an iterative solution. Note that the objective function (5) already incorporates all the constraints as discussed earlier. Hence, we only need to focus on the objective function itself.

When only considering a subset of application nodes $1, 2, ..., v_1 \leq V$, for a given mapping π, the value of the objective function for this subset of application nodes is

$$J_\pi(v_1) = \max \left\{ \max_{k,n} \sum_{v\leq v_1:\pi(v)=n} d_{v\to n,k}; \max_{(v-1,v)\in\mathcal{E}, v\leq v_1} c_{(v-1,v)\to(\pi(v-1),\pi(v))} \right\}. \tag{6}$$

Compared with (5), the only difference in (6) is that we consider the first v_1 application nodes and the mapping π is assumed to be given. The optimal cost for application nodes $1, 2, ..., v_1 \leq V$ is then

$$J_{\pi^*}(v_1) = \min_{\pi} J_\pi(v_1), \tag{7}$$

where π^* denotes the optimal mapping.

Proposition 1. ***(Decomposition of the Objective Function)**: Let $J_{\pi^*|\pi(v_1)}(v_1)$ denote the optimal cost under the condition that $\pi(v_1)$ is given, i.e. $J_{\pi^*|\pi(v_1)}(v_1) = \min_{\pi(1),...,\pi(v_1-1)} J_\pi(v_1)$ with given $\pi(v_1)$. When $\pi(v_1) = \pi(v_1-1) = ... = \pi(v_s) > \pi(v_s-1) \geq \pi(v_s-2) \geq ... \geq \pi(1)$, where $1 \leq v_s \leq v_1$, which means that v_s is mapped to a different physical node from $v_s - 1$ and nodes $v_s, ..., v_1$ are*

mapped onto the same physical node[4]*, then we have*

$$J_{\pi^*|\pi(v_1)}(v_1) = \min_{v_s=1,\dots,v_1} \min_{\pi(v_s-1)} \max \Bigg\{ J_{\pi^*|\pi(v_s-1)}(v_s-1); \max_{k=1,\dots,K} \sum_{v=v_s\dots v_1} d_{v\to\pi(v_1),k}; \max_{(v-1,v)\in\mathcal{E}, v_s\le v\le v_1} c_{(v-1,v)\to(\pi(v-1),\pi(v))} \Bigg\}. \quad (8)$$

The optimal mapping for v_1 can be found by

$$J_{\pi^*}(v_1) = \min_{\pi(v_1)} J_{\pi^*|\pi(v_1)}(v_1). \quad (9)$$

Proof. Because $\pi(v_s) = \pi(v_s+1) = ... = \pi(v_1)$, we have

$$J_\pi(v_1) = \max \Bigg\{ J_\pi(v_s-1); \max_{k=1,\dots,K} \sum_{v=v_s\dots v_1} d_{v\to\pi(v_1),k}; \max_{(v-1,v)\in\mathcal{E}, v_s\le v\le v_1} c_{(v-1,v)\to(\pi(v-1),\pi(v))} \Bigg\}. \quad (10)$$

The three terms in the maximum operation in (10) respectively correspond to: 1) the cost at physical nodes and edges that the application nodes $1,...,v_s-1$ (and their connecting edges) are mapped to, 2) the costs at the physical node that $v_s,...,v_1$ are mapped to, and 3) the pairwise costs for connecting v_s-1 and v_s as well as interconnections[5] of nodes in $v_s,...,v_1$. Taking the maximum of these three terms, we obtain the cost function in (6).

In the following, we focus on finding the optimal mapping based on the cost decomposition in (10). We note that the pairwise cost between v_s-1 and v_s depends on the placements of both v_s-1 and v_s. Therefore, in order to find the optimal $J_\pi(v_1)$ from $J_\pi(v_s-1)$, we need to find the minimum cost among all possible placements of v_s-1 and v_s, provided that nodes $v_s,...,v_1$ are mapped onto the same physical node and v_s and v_s-1 are mapped onto different physical nodes. For a given v_1, node v_s may be any node that satisfies $1 \le v_s \le v_1$. Therefore, we also need to search through all possible values of v_s. This can then be expressed as the proposition, where we first find $J_{\pi^*|\pi(v_1)}(v_1)$ as an intermediate step. □

Equation (8) is the Bellman's equation [38] for problem (5). Using dynamic programming [38], we can solve (5) by iteratively solving (8). In each iteration, the algorithm computes new costs $J_{\pi^*|\pi(v_1)}(v_1)$ for all possible mappings $\pi(v_1)$, based on the previously computed costs $J_{\pi^*|\pi(v)}(v)$ where $v < v_1$. For the final application node $v_1 = V$, we use (9) to compute the final optimal cost $J_{\pi^*}(V)$ and its corresponding mapping π^*.

[4]Note that when $v_s = 1$, then $v_s - 1$ does not exist, which means that all nodes $1,...,v_1$ are placed onto the same physical node. For convenience, we define $J_\pi(0) = 0$.

[5]Note that, although $v_s,...,v_1$ are mapped onto the same physical node, their pairwise costs may be non-zero if there exists additional cost when placing different application nodes onto the same physical node. In the extreme case where adjacent application nodes are not allowed to be placed onto the same physical node (i.e., conflict constraint), their pairwise cost when placing them on the same physical node becomes infinity.

Algorithm 1 Placement of a linear application graph onto a tree physical graph

1: Given linear application graph $\mathcal{R}$, tree physical graph $\mathcal{Y}$
2: Given $V \times N \times K$ matrix $\mathbf{D}$, its entries represent the weighted type k node cost $d_{v\to n,k}$
3: Given $(V-1) \times N \times N$ matrix $\mathbf{C}$, its entries represent the weighted pairwise cost $c_{(v-1,v)\to(n_1,n_2)}$
4: Define $V \times N$ matrix $\mathbf{J}$ to keep the costs $J_{\pi^*|\pi(v)=n}(v)$ for each node (v,n) in the auxiliary graph
5: Define $V \times N \times V$ matrix $\mathbf{\Pi}$ to keep the mapping corresponding to its cost $J_{\pi^*|\pi(v)=n}(v)$ for each node (v,n) in the auxiliary graph
6: **for** $v = 1...V$ **do**
7: **for** $n = 1...N$ **do**
8: Compute $J_{\pi^*|\pi(v)=n}(v)$ from (8), put the result into $\mathbf{J}$ and the corresponding mapping into $\mathbf{\Pi}$
9: **end for**
10: **end for**
11: Compute $J_{\pi^*}(V) \leftarrow \min_n J_{\pi^*|\pi(V)=n}(V)$
12: **return** the final mapping result π^* and final optimal cost $J_{\pi^*}(V)$

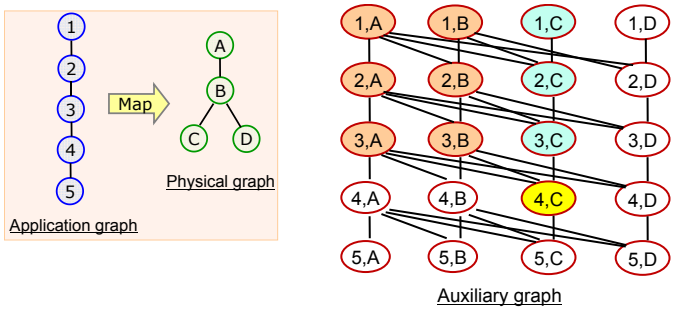

Figure 4. Auxiliary graph and algorithm procedure for the placement of a linear application graph onto a tree physical graph.

C. Optimal Algorithm

The pseudocode of the exact optimal algorithm is shown in Algorithm 1. It computes $V \cdot N$ number of $J_{\pi^*|\pi(v)=n}(v)$ values, and we take the minimum among no more than $V \cdot N$ values in (8). The terms in (8) include the sum or maximum of no more than V values and the maximum of K values. Because K is a constant in practical systems, we conclude that the *time-complexity of this algorithm is* $O(V^3N^2)$.

The *space-complexity of Algorithm 3 is* $O(VN(V+N))$, which is related to the memory required for storing matrices $\mathbf{D}$, $\mathbf{C}$, $\mathbf{J}$, and $\mathbf{\Pi}$ in the algorithm, where K is also regarded as a constant here.

Also note that the optimality of the result from Algorithm 1 is subject to the cycle-free constraint, and the sequence of nodes is always preserved in each iteration.

D. Example

To illustrate the procedure of the algorithm, we construct an auxiliary graph from the given application and physical graphs, as shown in Fig. 4. Each node (v_1, n_1) in the auxiliary graph

represents a possible placement of a particular application node, and is associated with the cost value $J_{\pi^*|\pi(v_1)=n_1}(v_1)$, where v_1 is the application node index and n_1 is the physical node index in the auxiliary graph. When computing the cost at a particular node, e.g. the cost $J_{\pi^*|\pi(4)=\mathrm{C}}(4)$ at node (4,C) in Fig. 4, the algorithm starts from the "earlier" costs $J_{\pi^*|\pi(v_s-1)}(v_s-1)$ where the tuple $(v_s-1, \pi(v_s-1))$ is either (1,A), (1,B), (2,A), (2,B), (3,A), or (3,B). From each of these nodes, the subsequent application nodes (i.e. from v_s to node 4) are all mapped onto physical node C, and we compute the cost for each such "path" with the maximum operations in (8), by assuming the values of v_s-1 and $\pi(v_s-1)$ are given by its originating node in the auxiliary graph. For example, one path can be (2,B) – (3,C) – (4,C) where $v_s-1=2$ and $\pi(v_s-1)$ = B, another path can be (1,A) – (2,C) – (3,C) – (4,C) where $v_s-1=1$ and $\pi(v_s-1)$ = A. Then, the algorithm takes the minimum of the costs for all paths, which corresponds to the minimum operations in (8) and gives $J_{\pi^*|\pi(4)=\mathrm{C}}(4)$. In the end, the algorithm searches through all the possible mappings for the final application node (node 5 in Fig. 4) and chooses the mapping that results in the minimum cost, which corresponds to the procedure in (9).

E. Extensions

The placement algorithm for single linear application graph can also be used when the objective function (in the form of (3) with $M=1$) is modified to one of the following:

$$\min_{\pi} \max \left\{ \max_{k,n} f_{n,k}\left(\sum_{v:\pi(v)=n} d_{v\to n,k} \right); \max_{l} g_l\left(\sum_{e=(v_1,v_2):(\pi(v_1),\pi(v_2))\ni l} b_{e\to l} \right) \right\}, \quad (11)$$

$$\min_{\pi} \left\{ \sum_{k,n} f_{n,k}\left(\sum_{v:\pi(v)=n} d_{v\to n,k} \right) + \sum_{l} g_l\left(\sum_{e=(v_1,v_2):(\pi(v_1),\pi(v_2))\ni l} b_{e\to l} \right) \right\}, \quad (12)$$

where $f_{n,k}(\cdot)$ and $g_l(\cdot)$ are increasing functions with $f_{n,k}(0)=0$, $g_l(0)=0$, $f_{n,k}(\infty)=\infty$, and $g_l(\infty)=\infty$. The algorithm and its derivation follow the same procedure as discussed above. These alternative objective functions can be useful for scenarios where the goal of optimization is other than min-max. The objective function in (12) will also be used later for solving the online placement problem.

IV. Online Placement Algorithms for Tree Application Graphs

Using the optimal algorithm for the single linear application graph placement as a sub-routine, we now present algorithms for the generalized case; namely, placement of an arriving stream of application graphs with tree topology. We first show that even the offline placement of a single tree is NP-hard. Then, we propose online algorithms to approximate the optimal placement with provable competitive ratio, by first considering the case where junction nodes in the application graph have pre-specified placements that are given beforehand, and later relax this assumption.

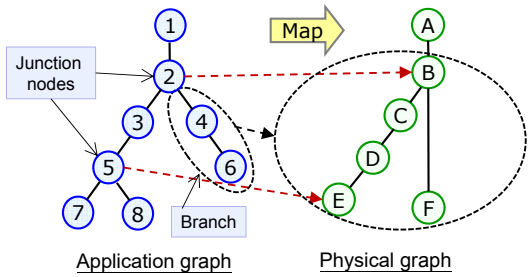

Figure 5. Example of application graph with given placement of junction nodes. Junction node 2 is placed on physical node B and junction node 5 is placed on physical node E. The algorithm needs to decide the placement of the remaining nodes, subject to the cycle-free constraint.

A. Hardness Result

Proposition 2. *(**NP-hardness**) Placement of a tree application graph onto a tree physical graph for the objective function defined in (3), with or without pre-specified junction node placement, is NP-hard.*

Proof. To show that the given problem is NP-hard, we show that the problem can be reduced from the NP-hard problem of minimum makespan scheduling on unrelated parallel machines (MMSUPM) [25], which minimizes the maximum load (or job processing time) on each machine.

Consider a special case in our problem where the application graph has a star topology with two levels (one root and multiple leaf nodes), and the physical graph is a line with multiple nodes. Assume that the number of resource types in the nodes is $K=1$, the application edge resource demand is zero, and the application node resource demand is non-zero. Then, the problem is essentially the MMSUPM problem. It follows that the MMSUPM problem reduces to our problem. In other words, if we can solve our problem in polynomial time, then we can also solve the MMSUPM problem in polynomial time. Because MMSUPM is NP-hard, our problem is also NP-hard. The above result holds no matter whether the root node (junction node) of the application graph has pre-specified placement or not. □

B. When All Junction Node Placements Are Given

We first consider tree application graphs for which the placements of junction nodes are given, and focus on placing the remaining non-junction nodes which are connected to at most two edges. An example is shown in Fig. 5. Given the placed junction nodes, we name the set of application edges and nodes that form a chain between the placed nodes (excluding each placed node itself, but including each edge that is connected to a placed node) as a *simple branch*, where the notion "simple" is opposed to the general branch which will be defined in Section IV-C. A simple branch can also be a chain starting from an edge that connects a placed node and ending at a leaf node, such as the nodes and edges within the

dashed boundary in the application graph in Fig. 5. Each node in a simple branch is connected to at most two edges.

1) Algorithm Design: We propose an online placement algorithm, where we borrow some ideas from [39]. Different from [39] which focused on routing and job scheduling problems, our work considers more general graph mapping.

When an application (represented by a tree application graph) arrives, we *split* the whole application graph into simple branches, and regard each simple branch as an independent application graph. All the nodes with given placement can also be regarded as an application that is placed before placing the individual simple branches. After placing those nodes, each individual simple branch is placed using the online algorithm that we describe below. In the remaining of this section, by application we refer to the *application after splitting*, i.e. each application either consists of a simple branch or a set of nodes with given placement.

How to Place Each Simple Branch: While our ultimate goal is to optimize (3), we use an *alternative objective function* to determine the placement of each newly arrived application i (after splitting). Such an indirect approach provides performance guarantee with respect to (3) in the long run. We will first introduce the new objective function and then discuss its relationship with the original objective function (3).

We define a variable $\hat{J}$ as a reference cost. The reference cost may be an estimate of the true optimal cost (defined as in (3)) from optimal offline placement, and we will discuss later about how to determine this value. Then, for placing the ith application, we use an objective function which has the same form as (12), with $f_{n,k}(\cdot)$ and $g_l(\cdot)$ defined as

$$f_{n,k}(x) \triangleq \exp_\alpha\left(\frac{p_{n,k}(i-1)+x}{\hat{J}}\right)-\exp_\alpha\left(\frac{p_{n,k}(i-1)}{\hat{J}}\right), \tag{13a}$$

$$g_l(x) \triangleq \exp_\alpha\left(\frac{p_l(i-1)+x}{\hat{J}}\right)-\exp_\alpha\left(\frac{p_l(i-1)}{\hat{J}}\right), \tag{13b}$$

subject to the cycle-free placement constraint, where we define $\exp_\alpha(y) \triangleq \alpha^y$ and $\alpha \triangleq 1+1/\gamma$ ($\gamma > 1$ is a design parameter).

Why We Use an Alternative Objective Function: The objective function (12) with (13a) and (13b) is the increment of the sum exponential values of the original costs, given all the previous placements. With this objective function, the performance bound of the algorithm can be shown analytically (see Proposition 3 below). Intuitively, the new objective function (12) serves the following purposes:

- "Guide" the system into a state such that the maximum cost among physical links and nodes is not too high, thus approximating the original objective function (3). This is because when the existing cost at a physical link or node (for a particular resource type k) is high, the incremental cost (following (12)) of placing the new application i on this link or node (for the same resource type k) is also high, due to the fact that $\exp_\alpha(y)$ is convex increasing and the cost definitions in (13a) and (13b).
- While (3) only considers the maximum cost, (12) is also related to the sum cost, because we sum up all the exponential cost values at different physical nodes and links together. This "encourages" a low resource consuming placement of the new application i (which is reflected by low sum cost), thus leaving more available resources for future applications. In contrast, if we use (3) directly for each newly arrived application, the placement may greedily take up too much resource, so that future applications can no longer be placed with a low cost.

In practice, we envision that objective functions with a shape similar to (12) can also serve our purpose.

Algorithm 2 Online placement of an application that is either a simple branch or a set of nodes with given placement

```
Given the ith application that is either a set of nodes with given placement or a simple branch
Given tree physical graph 𝒴
Given p_{n,k}(i−1), q_l(i−1), and placement costs
Given Ĵ and β
if application is a set of nodes with given placement then
    Obtain π_i based on given placement
else if application is a simple branch then
    Extend simple branch to linear graph 𝓡(i), by connecting zero-resource-demand application nodes to open edges, and the placements of these zero-resource-demand application nodes are given
    Run Algorithm 1 with objective function (12) with (13a) and (13b), for 𝓡(i), to obtain π_i
end if
if ∃n,k : p_{n,k}(i−1) + Σ_{v:π_i(v)=n} d_{v→n,k}(i) > βĴ or ∃l : q_l(i−1) + Σ_{e=(v_1,v_2):(π_i(v_1),π_i(v_2))∋l} b_{e→l}(i) > βĴ then
    return FAIL
else
    return π_i
end if
```

How to Solve It: Because each application either obeys a pre-specified placement or consists of a simple branch, we can use Algorithm 1 with appropriately modified cost functions to find the optimal solution to (12) with (13a) and (13b). For the case of a simple branch having an open edge, such as edge $(2,4)$ in Fig. 5, we connect an application node that has zero resource demand to extend the simple branch to a graph, so that Algorithm 1 is applicable.

Algorithm 2 summarizes the above argument as a formal algorithm for each application placement, where π_i denotes the mapping for the ith application. Define a parameter, $\beta = \log_\alpha\left(\frac{\gamma(NK+L)}{\gamma-1}\right)$, then Algorithm 2 performs the placement as long as the cost on each node and link is not bigger than $\beta\hat{J}$, otherwise it returns FAIL. The significance of the parameter β is in calculating the competitive ratio, i.e., the maximum ratio of the cost resulting from Algorithm 2 to the optimal cost from an equivalent offline placement, as shown below.

Why We Need the Reference Cost $\hat{J}$: The reference cost $\hat{J}$ is an input parameter of the objective function (12) and Algorithm 2, which enables us to show a performance bound for Algorithm 2, as shown in Proposition 3.

Proposition 3. *If there exists an offline mapping π^o that considers all M application graphs and brings cost J_{π^o}, such*

that $J_{\pi^o} \leq \hat{J}$, then Algorithm 2 never fails, i.e., $p_{n,k}(M)$ and $q_l(M)$ from Algorithm 2 never exceeds $\beta\hat{J}$. The cost J_{π^o} is defined in (3).

Proof. See Appendix B. □

Proposition 3 guarantees a bound for the cost resulting from Algorithm 2. We note that the optimal offline mapping π^{o*} produces cost $J_{\pi^{o*}}$, which is smaller than or equal to the cost of an arbitrary offline mapping. It follows that for any π^o, we have $J_{\pi^{o*}} \leq J_{\pi^o}$. This means that if there exists π^o such that $J_{\pi^o} \leq \hat{J}$, then we must have $J_{\pi^{o*}} \leq \hat{J}$. If we can set $\hat{J} = J_{\pi^{o*}}$, then from Proposition 3 we have $\max\{\max_{k,n} p_{n,k}(M); \max_l q_l(M)\} \leq \beta J_{\pi^{o*}}$, which means that the competitive ratio is β.

How to Determine the Reference Cost $\hat{J}$: Because the value of $J_{\pi^{o*}}$ is unknown, we cannot always set $\hat{J}$ exactly to $J_{\pi^{o*}}$. Instead, we need to set $\hat{J}$ to an estimated value that is not too far from $J_{\pi^{o*}}$. We achieve this by using the doubling technique, which is widely used in online approximation algorithms. The idea is to double the value of $\hat{J}$ every time Algorithm 2 fails. After each doubling, we ignore all the previous placements when calculating the objective function (12) with (13a) and (13b), i.e., we assume that there is no existing application, and we place the subsequent applications (including the one that has failed with previous value of $\hat{J}$) with the new value of $\hat{J}$. At initialization, the value of $\hat{J}$ is set to a reasonably small number $\hat{J}_0$.

In Algorithm 3, we summarize the high-level procedure that includes the splitting of the application graph, the calling of Algorithm 2, and the doubling process, with multiple application graphs that arrive over time.

2) Complexity and Competitive Ratio: In the following, we discuss the complexity and competitive ratio of Algorithm 3.

Because the value of $J_{\pi^{o*}}$ is finite[6], the doubling procedure in Algorithm 3 only contains finite steps. The remaining part of the algorithm mainly consists of calling Algorithm 2 which then calls Algorithm 1 for each simple branch. Because nodes and links in each simple branch together with the set of nodes with given placement add up to the whole application graph, similar to Algorithm 1, the *time-complexity of Algorithm 3 is* $O(V^3N^2)$ for each application graph arrival.

Similarly, when combining the procedures in Algorithms 1–3, we can see that the *space-complexity of Algorithm 3 is* $O(VN(V+N))$ for each application graph arrival, which is in the same order as Algorithm 1.

For the competitive ratio, we have the following result.

Proposition 4. ***(Competitive Ratio)**: Algorithm 3 is $4\beta = 4\log_\alpha\left(\frac{\gamma(NK+L)}{\gamma-1}\right)$-competitive.*

Proof. If Algorithm 2 fails, we know that $J_{\pi^{o*}} > \hat{J}$ according to Proposition 3. Hence, by doubling the value of $\hat{J}$ each time Algorithm 2 fails, we have $\hat{J}_f < 2J_{\pi^{o*}}$, where $\hat{J}_f$ is the final value of $\hat{J}$ after placing all M applications. Because we ignore all previous placements and only consider the applications $i_0, ..., i$ for a particular value of $\hat{J}$, it follows that

$$\max\ \Big\{\max_{k,n}\{p_{n,k}(i) - p_{n,k}(i_0-1)\}; \max_l\{q_l(i) - q_l(i_0-1)\}\Big\} \leq \beta\hat{J} \tag{14}$$

for the particular value of $\hat{J}$.

When we consider all the placements of M applications, by summing up (14) for all values of $\hat{J}$, we have

$$\max\left\{\max_{k,n} p_{n,k}(M); \max_l q_l(M)\right\} \leq \left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots\right)\beta\hat{J}_f < 2\left(1+\frac{1}{2}+\frac{1}{4}+\frac{1}{8}+\cdots\right)\beta J_{\pi^{o*}} = 4\beta J_{\pi^{o*}}.$$

Hence, the proposition follows. □

The variables α, γ and K are constants, and $L = N-1$ because the physical graph is a tree. Hence, the competitive ratio of Algorithm 3 can also be written as $O(\log N)$.

It is also worth noting that, for each application graph, we can have different tree physical graphs that are extracted from a general physical graph, and the above conclusions still hold.

Algorithm 3 High-level procedure for multiple arriving tree application graphs

```
Initialize Ĵ ← Ĵ_0
Define index i as the application index, which automatically increases by 1 for each new application (after splitting)
Initialize i ← 1
Initialize i_0 ← 1
loop
  if new application graph has arrived then
    Split the application graph into simple branches and a set of nodes with given placement, assume that each of them constitute an application
    for all application i do
      repeat
        Call Algorithm 2 for application i with p_{n,k}(i−1) = max{0, p_{n,k}(i−1) − p_{n,k}(i_0−1)} and q_l(i−1) = max{0, q_l(i−1) − q_l(i_0−1)}
        if Algorithm 2 returns FAIL then
          Set Ĵ ← 2Ĵ
          Set i_0 ← i
        end if
      until Algorithm 2 does not return FAIL
      Map application i according to π_i resulting from Algorithm 2
    end for
  end if
end loop
```

C. When at Least One Junction Node Placement Is Not Given

In this subsection, we focus on cases where the placements of some or all junction nodes are not given. For such scenarios,

[6]The value of $J_{\pi^{o*}}$ is finite unless the placement cost specification does not allow any placement with finite cost. We do not consider this case here because it means that the placement is not realizable under the said constraints. In practice, the algorithm can simply reject such application graphs when the mapping cost resulting from Algorithm 2 is infinity, regardless of what value of $\hat{J}$ has been chosen.

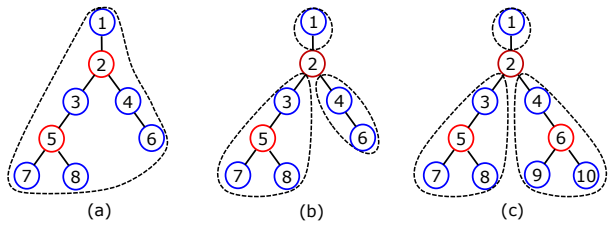

Figure 6. Example of application graphs with some unplaced junction nodes, the nodes and edges within each dashed boundary form a general branch: (a) nodes 2 and 5 are both unplaced, (b) node 2 is placed, node 5 is unplaced, (c) node 2 is placed, nodes 5 and 6 are unplaced.

we first extend our concept of branches to incorporate some unplaced junction nodes. The basic idea is that each *general branch* is the largest subset of nodes and edges that are interconnected with each other not including any of the nodes with pre-specified placement, but (as with our previous definition of simple branches) the subset includes the edges connected to placed nodes. A simple branch (see definition in Section IV-B) is always a general branch, but a general branch may or may not be a simple branch. Examples of general branches are shown in Fig. 6.

1) Algorithm Design: The main idea behind the algorithm is to combine Algorithm 2 with the enumeration of possible placements of unplaced junction nodes. When there is only a constant number of such nodes on any path from the root to a leaf, the algorithm remains polynomial in time-complexity while guaranteeing a polynomial-logarithmic (poly-log) competitive ratio.

To illustrate the intuition, consider the example application graph shown in Fig. 6(a), where nodes 2 and 5 are both initially unplaced. We follow a hierarchical determination of the placement of unplaced nodes starting with the nodes in the deepest level. For the example in Fig. 6(a), we first determine the placement of node 5, given each possible placement of node 2; then determine the placement of node 2. Recall that we use the cost function in (12) with (13a) and (13b) to determine the placement of each simple branch when all the junction nodes are placed. We use the same cost function (with slightly modified parameters) for the placement of nodes 2 and 5. However, when determining the placement of node 5, we regard the general branch that includes node 5 (which contains nodes 3, 5, 7, and 8 and the corresponding edges as shown in Fig. 6(b)) as one single application, i.e. the values of $p_{n,k}(i-1)$ and $q_l(i-1)$ in (13a) and (13b) correspond to the resource utilization at nodes and links before placing this whole general branch, and the application i contains all the nodes and edges in this general branch. Similarly, when determining the placement of node 2, we consider the whole application graph as a single application.

It is worth noting that in many cases we may not need to enumerate all the possible combinations of the placement of unplaced junction nodes. For example, in Fig. 6(c), when the placement of node 2 is given, the placement of nodes 5 and 6 does not impose additional restrictions upon each other (i.e., the placement of node 5 does not affect where node 6 can be placed, for instance). Hence, the general branches that respectively include node 5 and node 6 can be placed in a subsequent order using the online algorithm.

Algorithm 4 Tree-to-tree placement when some junction nodes are not placed

function Unplaced(v, h)
Given the ith application that is a general branch, tree physical graph $\mathcal{Y}$, $\hat{J}$, and β
Given $p_{n,k}(i-1)$ and $q_l(i-1)$ which is the current resource utilization on nodes and links
Define $\mathbf{\Pi}$ to keep the currently obtained mappings, its entry $\pi|_{\pi(v)=n_0}$ for all n_0 represents the mapping, given that v is mapped to n_0
Define $p_{n,k}(i)|_{\pi(v)=n_0}$ and $q_l(i)|_{\pi(v)=n_0}$ for all n_0 as the resource utilization after placing the ith application, given that v is mapped to n_0
Initialize $p_{n,k}(i)|_{\pi(v)=n_0} \leftarrow p_{n,k}(i-1)$ and $q_l(i)|_{\pi(v)=n_0} \leftarrow q_l(i-1)$ for all n_0
for all n_0 that v can be mapped to **do**
  Assume v is placed at n_0
  for all general branch that is connected with v **do**
    if the general branch contains unplaced junction nodes **then**
      Find the top-most unplaced vertex v' within this general branch
      Call Unplaced($v', h-1$) while assuming v is placed at n_0, and with $p_{n,k}(i-1) = p_{n,k}(i)|_{\pi(v)=n_0}$ and $q_l(i-1) = q_l(i)|_{\pi(v)=n_0}$
    else
      (in which case the general branch is a simple branch without unplaced junction nodes)
      Run Algorithm 2 for this branch
    end if
    Put mappings resulting from Unplaced($v', h-1$) or Algorithm 2 into $\pi|_{\pi(v)=n_0}$
    Update $p_{n,k}(i)|_{\pi(v)=n_0}$ and $q_l(i)|_{\pi(v)=n_0}$ to incorporate new mappings
  end for
end for
Find $\min_{n_0} \sum_{k,n}\left(\exp_\alpha\left(\frac{p_{n,k}(i)|_{\pi(v)=n_0}}{\beta^h \hat{J}}\right) - \exp_\alpha\left(\frac{p_{n,k}(i-1)}{\beta^h \hat{J}}\right)\right) + \sum_l \left(\exp_\alpha\left(\frac{q_l(i)|_{\pi(v)=n_0}}{\beta^h \hat{J}}\right) - \exp_\alpha\left(\frac{q_l(i-1)}{\beta^h \hat{J}}\right)\right)$, returning the optimal placement of v as n_0^*.
if $h = H$ and ($\exists n,k : p_{n,k}(i)|_{\pi(v)=n_0^*} > \beta^{1+H}\hat{J}$ or $\exists l : q_l(i)|_{\pi(v)=n_0^*} > \beta^{1+H}\hat{J}$) **then**
  return FAIL
else
  return $\pi|_{\pi(v)=n_0^*}$
end if

Based on the above examples, we summarize the procedure as Algorithm 4, where we solve the problem recursively and determine the placement of one junction node that has not been placed before in each instance of the function Unplaced(v, h). The parameter v is initially set to the top-most unplaced junction node (node 2 in Fig. 6(a)), and h is initially set to H (the maximum number of unplaced junction nodes on any path from the root to a leaf in the application graph).

Algorithm 4 can be embedded into Algorithm 3 to handle multiple arriving application graphs and unknown reference cost $\hat{J}$. The only part that needs to be modified in Algorithm 3 is that it now splits the whole application graph into general branches (rather than simple branches without unplaced junction nodes), and it either calls Algorithm 2 or Algorithm 4 depending on whether there are unplaced junction nodes in the corresponding general branch. When there are such nodes, it calls Unplaced(v, h) with the aforementioned initialization parameters.

2) Complexity and Competitive Ratio: The *time-complexity of Algorithm 4* together with its high-level procedure that is a modified version of Algorithm 3 is $O(V^3N^{2+H})$ for each application graph arrival, as explained below. Note that H is generally not the total number of unplaced nodes.

Obviously, when $H = 0$, the time-complexity is the same as the case where all junction nodes are placed beforehand. When there is only one unplaced junction node (in which case $H = 1$), Algorithm 4 considers all possible placements for this vertex, which has at most N choices. Hence, its time-complexity becomes N times the time-complexity with all placed junction nodes. When there are multiple unplaced junction nodes, we can see from Algorithm 4 that it only increases its recursion depth when some lower level unplaced junction nodes exist. In other words, parallel general branches (such as the two general branches that respectively include node 5 and node 6 in Fig. 6(c)) do not increase the recursion depth, because the function Unplaced(v, h) for these general branches is called in a sequential order. Therefore, the time-complexity depends on the maximum recursion depth which is H; thus, the overall time-complexity is $O(V^3N^{2+H})$.

The *space-complexity of Algorithm 4 is* $O(VN^{1+H}(V+N))$ for each application graph arrival, because in every recursion, the results for all possible placements of v are stored, and there are at most N such placements for each junction node.

Regarding the competitive ratio, similar to Proposition 3, we can obtain the following result.

Proposition 5. *If there exists an offline mapping π^o that considers all M application graphs and brings cost J_{π^o}, such that $J_{\pi^o} \leq \hat{J}$, then Algorithm 4 never fails, i.e., $p_{n,k}(M)$ and $q_l(M)$ resulting from Algorithm 4 never exceeds $\beta^{1+H}\hat{J}$.*

Proof. When $H = 0$, the claim is the same as Proposition 3. When $H = 1$, there is at most one unplaced junction node in each general branch. Because Algorithm 4 operates on each general branch, we can regard that we have only one unplaced junction node when running Algorithm 4. In this case, there is no recursive calling of Unplaced(v, h). Recall that v is the top-most unplaced junction node. The function Unplaced(v, h) first fixes the placement of v to a particular physical node n_0, and finds the placement of the remaining nodes excluding v. It then finds the placement of v.

From Proposition 3, we know that when we fix the placement of v, the cost resulting from the algorithm never exceeds $\beta\hat{J}$ if there exists a mapping $\pi^o|_{\pi(v)=n_0}$ (under the constraint that v is placed at n_0) that brings cost $J_{\pi^o|_{\pi(v)=n_0}} \leq \hat{J}$.

To find the placement of v, Algorithm 4 finds the minimum cost placement from the set of placements that have been obtained when the placement of v is given. Reapplying Proposition 3 for the placement of v, by substituting $\hat{J}$ with $\beta\hat{J}$, we know that the cost from the algorithm never exceeds $\beta^2\hat{J}$, provided that there exists a mapping, which is within the set of mappings produced by the algorithm with given v placements[7], that has a cost not exceeding $\beta\hat{J}$. Such a mapping exists and can be produced by the algorithm if there exists an offline mapping π^o (thus a mapping $\pi^o|_{\pi(v)=n_0}$ for a particular placement of v) that brings cost J_{π^o} with $J_{\pi^o} \leq \hat{J}$. Hence, the claim follows for $H = 1$.

When $H > 1$, because we decrease the value of h by one every time we recursively call Unplaced(v, h), the same propagation principle of the bound applies as for the case with $H = 1$. Hence, the claim follows. □

Using the same reasoning as for Proposition 4, it follows that *Algorithm 4 in combination with the extended version of Algorithm 3 is* $4\beta^{1+H} = 4\log_\alpha^{1+H}\left(\frac{\gamma(NK+L)}{\gamma-1}\right)$*-competitive, thus its competitive ratio is* $O(\log^{1+H} N)$.

V. Numerical Evaluation

We compare the proposed algorithm against two heuristic approaches via simulation. The first approach is one that greedily minimizes the maximum resource utilization (according to (3)) for the placement of *every* newly arrived application graph. The second approach is the Vineyard algorithm proposed in [27], where load balancing is also considered as a main goal in application placement.

Both the greedy and Vineyard algorithms require an optimization problem to be solved as a subroutine, for the placement of every newly arrived application. This optimization problem can be expressed as a mixed-integer linear program (MILP). MILPs are generally *not* solvable in polynomial-time, thus an LP-relaxation and rounding procedure is used in [27]. In this paper, to capture the best generality and eliminate inaccuracies caused by heuristic rounding mechanisms (because there are multiple ways of rounding that one could use), we solve the MILP subroutines directly using CPLEX [40]. This gives an exact solution to the subroutine, thus the greedy and Vineyard algorithms in the simulation may perform better than they would in reality, and we are conservative in showing the effectiveness of the proposed algorithm.

Note that these MILP solutions do *not* represent the optimal offline solution, because an optimal offline solution needs to consider all application graphs at the same time, whereas the methods that we use for comparison only solve the MILP subroutine for each newly arrived application. Obtaining the optimal offline solution requires excessive computational time such that the simulation infeasible, hence we do not consider

[7]Note that, as shown in Line 20 of Algorithm 4, to determine the placement of v, we only take the minimum cost (expressed as the difference of exponential functions) with respect to those mappings that were obtained with given placement of v. It follows that the minimization is only taken among a subset of all the possible mappings. This restricts the reference mapping to be within the set of mappings that the minimization operator operates on. Because, only in this way, the inequality (19) in the proof of Proposition 3 can be satisfied. On the contrary, Algorithm 2 considers all possible mappings that a particular simple branch can be mapped to, by calling Algorithm 1 as its subroutine.

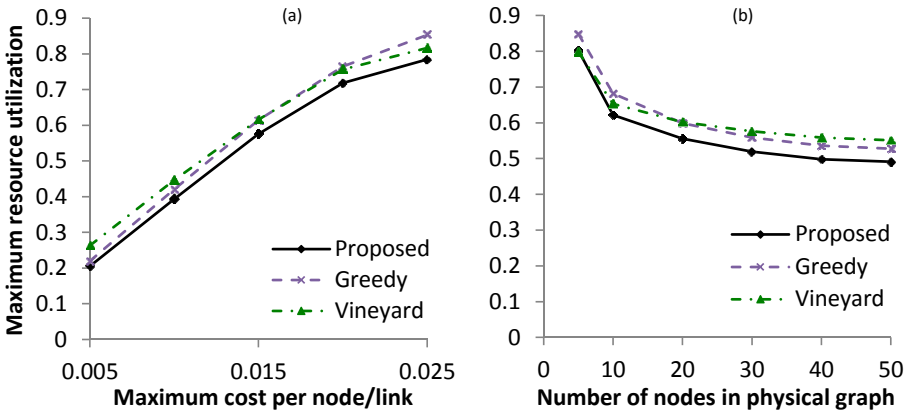

Figure 7. Maximum resource utilization when junction node placements are pre-specified.

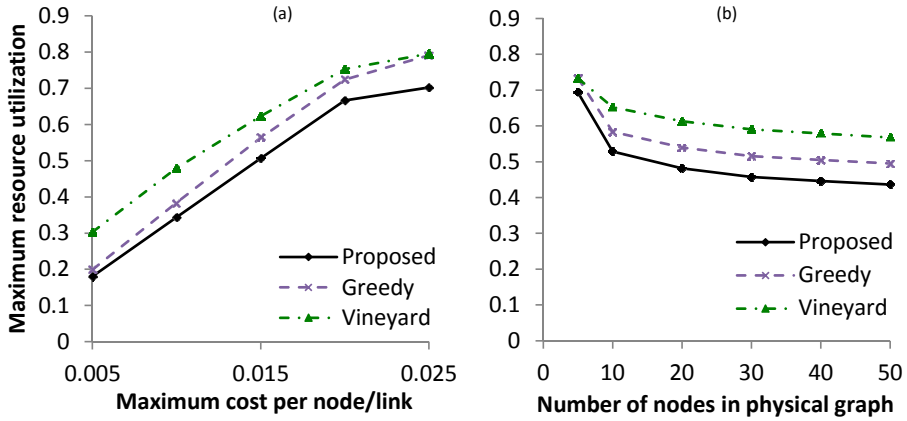

Figure 8. Maximum resource utilization when junction node placements are not pre-specified.

it here. We also do not compare against the theoretical approach in [28] via simulation, because that approach is non-straightforward to implement. However, we have outlined the benefits of our approach against [28] in Section I-C and some further discussion will be given in Section VI.

To take into account possible negative impacts of the cycle-free restriction in the proposed algorithm, we do *not* impose the cycle-free constraint in the baseline greedy and Vineyard algorithms. However, for a fair comparison, we do require in the baseline approaches that when the placements of junction nodes are given, the children of this junction node can only be placed onto the physical node on which the junction node has been placed, or onto the children of this physical node.

Because MEC is a very new concept which has not been practically deployed in a reasonably large scale, we currently do not have real topologies available to us for evaluation. Therefore, similar to existing work such as [27], we consider synthetic tree application and physical graphs. Such graphs mimic realistic MEC setups where MEC servers and applications locate at multiple network equipments in different hierarchical levels, see [8] and the example in Section I for instance. The number of application nodes for each application is randomly chosen from the interval $[3, 10]$, and the number of physical nodes ranges from 2 to 50. This simulation setting is similar to that in [27]. We use a sequential approach to assign connections between nodes. Namely, we first label the nodes with indices. Then, we start from the lowest index, and connect each node m to those nodes that have indices $1, 2, ..., m-1$. Node m connects to node $m-1$ with probability 0.7, and connects to nodes $1, 2, ..., m-2$ each with probability $0.3/(m-2)$. We restrict the application root node to be placed onto the physical root node, considering that some portion of processing has to be performed on the core cloud possibly due to the constraint of database location (see Fig. 1). We consider 100 application arrivals and simulate with 100 different random seeds to obtain the overall performance. The placement cost of a single node or link is uniformly distributed between 0 and a maximum cost. For the root application node, the cost is divided by a factor of 10. We set the design parameter $\gamma = 2$.

Figures 7 and 8 show the maximum resource utilization, i.e., the value of (3), averaged over results from different random seeds[8], respectively with and without pre-specified placement of junction nodes. In Figs. 7(a) and 8(a), the

[8]We only consider those random seeds which produce a maximum resource utilization that is smaller than one, because otherwise, the physical network is considered as overloaded after hosting 100 applications. We also observed in the simulations that the number of accepted applications is similar when using different methods. The relative degradation in the number of accepted applications of the proposed method compared with other methods never exceeds 2% in the simulations.

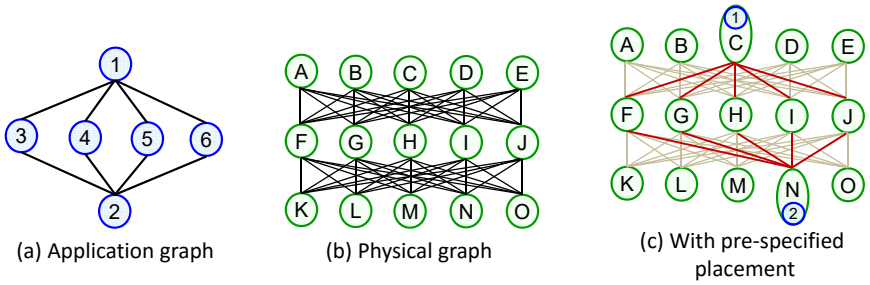

Figure 9. Example where application and physical graphs are not trees: (a) application graph, (b) physical graph, (c) restricted physical graph with pre-specified placement of application nodes 1 and 2.

number of physical nodes is randomly chosen from the interval $[2, 50]$; and in Figs. 7(b) and 8(b), the maximum cost per application node/link is set to 0.015. It is evident that the proposed method outperforms those methods in comparison. The resource utilization tends to converge when the number of physical nodes is large because of the fixed root placement. As mentioned earlier, practical versions of greedy and Vineyard algorithms that have LP-relaxation and rounding may perform worse than what our current results show.

We now explain why the proposed method outperforms other methods. We first note that the uniqueness in the proposed algorithm is that it uses a non-linear objective function for placing each new application, whereas the baseline methods and most other existing approaches use linear objective functions. The exponential-difference cost (12) with (13a) and (13b) used in the proposed algorithm for the placement of each newly arrived application graph aims at both load balancing and reducing sum resource utilization. It leaves more space for applications that arrive in the future. Therefore, it outperforms the greedy approach which does not take future arrivals into account. The Vineyard approach does not strongly enforce load balancing unless operating close to the resource saturation point, due to the characteristics of its objective function used in each subroutine of application arrival.

When comparing Fig. 7 to Fig. 8, we can find that the performance gaps between the proposed method and other methods are larger when the junction nodes are not placed beforehand. This is mainly because the judgment of whether Algorithm 4 has failed is based on the factor β^{1+H}, and for Algorithm 2 it is based on β. It follows that Algorithm 4 is less likely to fail when $H > 0$. In this case, the value of $\hat{J}$ is generally set to a smaller value by the doubling procedure in Algorithm 3. A smaller value of $\hat{J}$ also results in a larger change in the exponential-difference cost when the amount of existing load changes[9]. This brings a better load balancing on average (but not for the worst-case, the worst-case result is still bounded by the bounds derived earlier in this paper).

VI. Discussion

Is the Tree Assumption Needed? For ease of presentation and considering the practical relevance to MEC applications, we have focused on tree-to-tree placements in this paper. However, the tree assumption is *not* absolutely necessary for our algorithms to be applicable. For example, consider the placement problem shown in Fig. 9, where the application graph contains two junction nodes[10] (nodes 1 and 2) and multiple simple branches (respectively including nodes 3, 4, 5, and 6) between these two junction nodes. Such an application graph is common in applications where processing can be parallelized at some stage. The physical graph shown in Fig. 9(b) still has a hierarchy, but we now have connections between all pairs of nodes at two adjacent levels. Obviously, neither the application nor the physical graph in this problem has a tree structure.

Let us assume that junction node 1 has to be placed at the top level of the physical graph (containing nodes A, B, C, D, E), junction node 2 has to be placed at the bottom level of the physical graph (containing nodes K, L, M, N, O), and application nodes $3, 4, 5, 6$ have to be placed at the middle level of the physical graph (containing nodes F, G, H, I, J). One possible junction node placement under this restriction is shown in Fig. 9(c). With this pre-specified junction node placement, the mapping of each application node in $\{3, 4, 5, 6\}$ can be found by the simple branch placement algorithm (Algorithm 3 which embeds Algorithm 2) introduced earlier, because it only needs to map each application node in $\{3, 4, 5, 6\}$ onto each physical node in {F, G, H, I, J}, and find the particular assignment that minimizes (12) with (13a) and (13b). Therefore, in this example, when the junction node placements are pre-specified, the proposed algorithm can find the placement of other application nodes with $O(V^3N^2)$ time-complexity, which is the complexity of Algorithm 3 as discussed in Section IV-B2. When the junction node placements are not pre-specified, the proposed algorithm can find the placement of the whole application graph with $O(V^3N^4)$ time-complexity, because here $H = 2$ (recall that the complexity result was derived in Section IV-C2).

We envision that this example can be generalized to a class of application and physical graphs where there exist a limited number of junction nodes that are not placed beforehand. The algorithms proposed in this paper should still be applicable to such cases, as long as we can find a limited number of cycle-free paths between two junction nodes when they are placed on the physical graph. We leave a detailed discussion on this

[9]This is except for the top-level instance of Unplaced(v, h) due to the division by β^h in Line 20 of Algorithm 4.

[10]For non-tree graphs, a junction node can be defined as those nodes that are not part of a simple branch.

aspect as future work.

Practical Implications: Besides the proposed algorithms themselves, the results of this paper also reveal the following insights that may guide future implementation:

1) The placement is easier when the junction nodes are placed beforehand. This is obvious when comparing the time-complexities and competitive ratios for cases with and without unplaced junction nodes.
2) There is a trade-off between instantaneously satisfying the objective function and leaving more available resources for future applications. Leaving more available resources may cause the system to operate in a sub-optimal state for the short-term, but future applications may benefit from it. This trade-off can be controlled by defining an alternative objective function which is different from (but related to) the overall objective that the system tries to achieve (see Section IV-B1).

Performance Bound Comparison: As mentioned in Section I, [28] is the only work which we know that has studied the competitive ratio of online application placement considering both node and link optimization. Our approach has several benefits compared to [28] as discussed in Section I-C. Besides those benefits, we would like to note that the proposed algorithm outperforms [28] in time-complexity, space-complexity, and competitive ratio when the placements of all junction nodes (if any) are pre-specified. The performance bounds of these two approaches can be found in Sections I-B and I-D, respectively. Note that a linear application graph does not have any junction node, thus it falls into the above category. Linear application graphs are the case for a typical class of MEC applications (see the example in Section I-A) as well as for related problems such as service chain embedding [32]–[34]. When some junction node placements are not pre-specified, our approach provides a performance bound comparable to that in [28], because $H \leq D$. Moreover, [28] does not provide exact optimal solutions for the placement of a single linear application graph; it also does not have simulations to show the average performance of the algorithm.

Tightness of Competitive Ratio: By comparing the competitive ratio result of our approach to that in [28], we see that both approaches provide poly-log competitive ratios for the general case. It is however unclear whether this is the best performance bound one can achieve for the application placement problem. This is an interesting but difficult aspect worth studying in the future.

VII. Conclusions

In this paper, the placement of an incoming stream of application graphs onto a physical graph has been studied under the MEC context. We have first proposed an exact optimal algorithm for placing one linear application graph onto a tree physical graph which works for a variety of objective functions. Then, with the goal of minimizing the maximum resource utilization at physical nodes and links, we have proposed online approximation algorithms for placing tree application graphs onto tree physical graphs. When the maximum number of unplaced junction nodes on any path from the root to a leaf (in the application graph) is a constant, the proposed algorithm has polynomial time and space complexity and provides poly-log worst-case optimality bound (i.e., competitive ratio). Besides the theoretical evaluation of worst-case performance, we have also shown the average performance via simulation. A combination of these results implies that the proposed method performs reasonably well on average and it is also robust in extreme cases.

The results in this paper can be regarded as an initial step towards a more comprehensive study in this direction. Many constraints in the problem formulation are for ease of presentation, and can be readily relaxed for a more general problem. For example, as discussed in Section VI, the tree-topology restriction is not absolutely essential for the applicability of our proposed algorithms. The algorithms also work for a class of more general graphs as long as the cycle-free constraint is satisfied. While we have not considered applications leaving at some time after their arrival, our algorithm can be extended to incorporate such cases, for example using the idea in [41]. The algorithm for cases with unplaced junction nodes is essentially considering the scenario where there exists some low-level placement (for each of the branches) followed by some high level placement (for the junction nodes). Such ideas may also be useful in developing practical distributed algorithms with provable performance guarantees.

Appendix A
Approximation Ratio for Cycle-free Mapping

We focus on how well the cycle-free restriction approximates the more general case which allows cycles, for the placement of a single linear application graph. We first show that with the objective of load balancing (defined in (3) in Section II-B), the problem of placing a single linear application graph onto a linear physical graph when allowing cycles is NP-hard, and then discuss the approximation ratio of the cycle-free restriction.

Proposition 6. *The line-to-line placement problem for the objective function defined in (3)* while allowing cycles *is NP-hard.*

Proof. The proof is similar with the proof of Proposition 2 in Section IV-A, namely the problem can be reduced from the minimum makespan scheduling on unrelated parallel machines (MMSUPM) problem. Consider the special case where the edge demand is zero, then the problem is the same with the MMSUPM problem, which deals with placing V jobs onto N machines without restriction on their ordering, with the goal of minimizing the maximum load on each machine. □

To discuss the approximation ratio of the cycle-free assignment, we separately consider edge costs and node costs. The worst case ratio is then the maximum among these two ratios, because we have $\max\{r_1x_1, r_2x_2\} \leq \max\{r_1, r_2\} \cdot \max\{x_1, x_2\}$, for arbitrary $r_1, r_2, x_1, x_2 \geq 0$. The variables x_1 and x_2 can respectively denote the true optimal maximum costs at nodes and links, and the variables r_1 and r_2 can be their corresponding approximation ratios. Then, $\max\{x_1, x_2\}$ is the true optimal maximum cost when considering nodes and

links together, and $\max\{r_1, r_2\}$ is their joint approximation ratio. The joint approximation ratio $\max\{r_1, r_2\}$ is tight (i.e., there exists a problem instance where the actual optimality gap is arbitrarily close the approximation ratio, recall that the approximation ratio is defined in an upper bound sense) when r_1 and r_2 are tight, because we can construct worst-case examples, one with zero node demand and another with zero link demand, and there must exist one worst-case example which has approximation ratio $\max\{r_1, r_2\}$.

In the following discussion, we assume that the application and physical nodes are indexed in the way described in Section III-A. The following proposition shows that cycle-free placement is always optimal when only the edge cost is considered.

Proposition 7. *Cycle-free placement on tree physical graphs always has lower or equal maximum edge cost compared with placement that allows cycles.*

Proof. Suppose a placement that contains cycles produces a lower maximum edge cost than any cycle-free placement, then there exists v and v_1 $(v_1 > v+1)$ both placed on a particular node n, while nodes $v+1, ..., v_1-1$ are placed on some nodes among $n+1, ..., N$. In this case, placing nodes $v+1, ..., v_1-1$ all onto node n never increases the maximum edge cost, which shows a contradiction. □

For the node cost, we first consider the case where the physical graph is a single line. We note that in this case the cycle-free placement essentially becomes an "ordered matching", which matches V items into N bins, where the first bin may contain items $1, ..., v_1$, the second bin may contain items $v_1+1, ..., v_2$, and so on. We can also view the problem as partitioning the ordered set $\mathcal{V}$ into N subsets, and each subset contains consecutive elements from $\mathcal{V}$.

Proposition 8. *When each application node has the same cost when placing it on any physical node, then the cycle-free line-to-line placement has a* tight *approximation ratio of 2.*

Proof. Suppose we have V items that can be packed into N bins by a true optimal algorithm (which does not impose ordering on items), and the optimal cost at each bin is OPT.

To show that the worst case cost ratio resulting from the ordering cannot be larger than 2, we consider a bin packing where the size of each bin is OPT. (Note that the bin packing problem focuses on minimizing the number of bins with given bin size, which is slightly different from our problem.) Because an optimal solution can pack our V items into N bins with maximum cost OPT, when we are given that the size of each bin is OPT, we can also pack all the V items into N bins. Hence, the optimal solution to the related bin packing problem is N. When we have an ordering, we can do the bin packing by the first-fit algorithm which preserves our ordering. The result of the first-fit algorithm has been proven to be at most $2N$ bins [25].

Now we can combine two neighboring bins into one bin. Because we have at most $2N$ bins from the first-fit algorithm, we will have at most N bins after combination. Also because each bin has size OPT in the bin packing problem, the cost after combination will be at most $2 \cdot \mathrm{OPT}$ for each bin. This shows that the worst-case cost for ordered items is at most $2 \cdot \mathrm{OPT}$.

To show that the approximation ratio of 2 is tight, we consider the following problem instance as a tight example. Suppose $V = 2N$. Among the $2N$ items, N of them have cost of $(1-\epsilon)\cdot\mathrm{OPT}$, where $\epsilon > \frac{1}{1+N}$, the remaining N have a cost of $\epsilon \cdot \mathrm{OPT}$. Obviously, an optimal allocation will put one $(1-\epsilon)\cdot\mathrm{OPT}$ item and one $\epsilon\cdot\mathrm{OPT}$ item into one bin, and the resulting maximum cost at each bin is OPT.

A bad ordering could have all $(1-\epsilon)\cdot\mathrm{OPT}$ items coming first, and all $\epsilon\cdot\mathrm{OPT}$ items coming afterwards. In this case, if the ordered placement would like the maximum cost to be smaller than $(2-2\epsilon)\cdot\mathrm{OPT}$, it would be impossible to fit all the items into N bins, because all the $(1-\epsilon)\cdot\mathrm{OPT}$ items will already occupy N bins, as it is impossible to put more than one $(1-\epsilon)\cdot\mathrm{OPT}$ item into each bin if the cost is smaller than $(2-2\epsilon)\cdot\mathrm{OPT}$. Because $N\epsilon\cdot\mathrm{OPT} > \left(\frac{1}{\epsilon}-1\right)\epsilon\cdot\mathrm{OPT} = (1-\epsilon)\cdot\mathrm{OPT}$, it is also impossible to put all $\epsilon\cdot\mathrm{OPT}$ into the last bin on top of the existing $(1-\epsilon)\cdot\mathrm{OPT}$ item. This means an ordered placement of these V items into N bins has a cost that is at least $(2-2\epsilon)\cdot\mathrm{OPT}$

Considering arbitrarily large N and thus arbitrarily small ϵ, we can conclude that the approximation ratio of $2\cdot\mathrm{OPT}$ is tight. □

Corollary 1. *When the physical graph is a tree and the maximum to minimum cost ratio for placing application node v on any physical node is $d_{\%,v}$, then the cycle-free line-to-line placement has an approximation ratio of $2V \cdot \max_v d_{\%,v} = O(V)$.*

Proof. This follows from the fact that OPT may choose the minimum cost for each v while the ordered assignment may have to choose the maximum cost for some v, and also, in the worst case, the cycle-free placement may place all application nodes onto one physical node. The factor 2 follows from Proposition 8. □

It is not straightforward to find out whether the bound in the above corollary is tight or not, thus we do not discuss it here.

We conclude that the cycle-free placement always brings optimal link cost, which is advantageous. The approximation ratio of node costs can be $O(V)$ in some extreme cases. However, the cycle-free restriction is still reasonable in many practical scenarios. Basically, in these scenarios, one cannot split the whole workload onto all the available servers without considering the total link resource consumption. The analysis here is also aimed to provide some further insights that helps to justify in what practical scenarios the proposed work is applicable, while further study is worthwhile for some other scenarios.

APPENDIX B
PROOF OF PROPOSITION 3

The proof presented here borrows ideas from [39], but is applied here to the generalized case of graph mappings and arbitrary reference offline costs J_{π^o}. For a given $\hat{J}$,

we define $\tilde{p}_{n,k}(i) = p_{n,k}(i)/\hat{J}$, $\tilde{d}_{v\to n,k}(i) = d_{v\to n,k}(i)/\hat{J}$, $\tilde{q}_l(i) = q_l(i)/\hat{J}$, and $\tilde{b}_{e\to l}(i) = b_{e\to l}(i)/\hat{J}$. To simplify the proof structure, we first introduce some notations so that the link and node costs can be considered in an identical framework, because it is not necessary to distinguish them in the proof of this proposition. We refer to each type of resources as an *element*, i.e., the type k resource at node n is an element, the resource at link l is also an element. Then, we define the aggregated cost up to application i for element r as $\tilde{z}_r(i)$. The value of $\tilde{z}_r(i)$ can be either $\tilde{p}_{n,k}(i)$ or $\tilde{q}_l(i)$ depending on the resource type under consideration. Similarly, we define $\tilde{w}_{r|\pi}(i)$ as the incremental cost that application i brings to element r under the mapping π. The value of $\tilde{w}_{r|\pi}(i)$ can be either $\sum_{\forall v:\pi(v)=n} \tilde{d}_{v\to n,k}(i)$ or $\sum_{\forall e=(v_1,v_2):(\pi(v_1),\pi(v_2))\ni l} \tilde{b}_{e\to l}(i)$. Note that both $\tilde{z}_r(i)$ and $\tilde{w}_{r|\pi}(i)$ are normalized by the reference cost $\hat{J}$.

Using the above notations, the objective function in (12) with (13a) and (13b) becomes

$$\min_{\pi_i} \sum_r \left(\alpha^{\tilde{z}_r(i-1)+\tilde{w}_{r|\pi_i}(i)} - \alpha^{\tilde{z}_r(i-1)}\right). \tag{15}$$

Note that due to the notational equivalence, (15) is the same as (12) with (13a) and (13b).

Recall that π^o denotes the reference offline mapping result, let π_i^o denote the offline mapping result for nodes that correspond to the ith application, and $\tilde{z}_r^o(i)$ denote the corresponding aggregated cost until application i. Define the following potential function:

$$\Phi(i) = \sum_r \alpha^{\tilde{z}_r(i)} \left(\gamma - \tilde{z}_r^o(i)\right), \tag{16}$$

which helps us prove the proposition. Note that variables without superscript "o" correspond to the values resulting from Algorithm 2 that optimizes the objective function (15) for each application independently.

The change in $\Phi(i)$ after new application arrival is

$$\begin{aligned}
&\Phi(i) - \Phi(i-1)\\
&= \sum_{r:\exists\pi_i(\cdot)=r} \left(\alpha^{\tilde{z}_r(i)} - \alpha^{\tilde{z}_r(i-1)}\right)\left(\gamma - \tilde{z}_r^o(i-1)\right)\\
&\quad - \sum_{r:\exists\pi_i^o(\cdot)=r} \alpha^{\tilde{z}_r(i)}\tilde{w}_{r|\pi_i^o}(i) &(17)\\
&\le \sum_{r:\exists\pi_i(\cdot)=r} \gamma\left(\alpha^{\tilde{z}_r(i-1)+\tilde{w}_{r|\pi_i}(i)} - \alpha^{\tilde{z}_r(i-1)}\right)\\
&\quad - \sum_{r:\exists\pi_i^o(\cdot)=r} \alpha^{\tilde{z}_r(i-1)}\tilde{w}_{r|\pi_i^o}(i) &(18)\\
&\le \sum_{r:\exists\pi_i^o(\cdot)=r} \gamma\left(\alpha^{\tilde{z}_r(i-1)+\tilde{w}_{r|\pi_i^o}(i)} - \alpha^{\tilde{z}_r(i-1)}\right)\\
&\quad - \alpha^{\tilde{z}_r(i-1)}\tilde{w}_{r|\pi_i^o}(i) &(19)\\
&= \sum_{r:\exists\pi_i^o(\cdot)=r} \alpha^{\tilde{z}_r(i-1)}\left\{\gamma\left(\alpha^{\tilde{w}_{r|\pi_i^o}(i)} - 1\right) - \tilde{w}_{r|\pi_i^o}(i)\right\}, &(20)
\end{aligned}$$

where the notation $\pi_i(\cdot) = r$ or $\pi_i^o(\cdot) = r$ means that application i has occupied some resource from element r when respectively using the mapping from Algorithm 2 or the reference offline mapping.

We explain the relationships in (17)–(20) in the following. Equality (17) follows from

$$\begin{aligned}
&\Phi(i) - \Phi(i-1)\\
&= \sum_r \alpha^{\tilde{z}_r(i)}\left(\gamma - \left(\tilde{z}_r^o(i-1) + \tilde{w}_{r|\pi_i^o}(i)\right)\right)\\
&\quad - \sum_r \alpha^{\tilde{z}_r(i-1)}\left(\gamma - \tilde{z}_r^o(i-1)\right)\\
&= \sum_r \left(\alpha^{\tilde{z}_r(i)} - \alpha^{\tilde{z}_r(i-1)}\right)\left(\gamma - \tilde{z}_r^o(i-1)\right)\\
&\quad - \sum_r \alpha^{\tilde{z}_r(i)}\tilde{w}_{r|\pi_i^o}(i)\\
&= \sum_{r:\exists\pi_i(\cdot)=r} \left(\alpha^{\tilde{z}_r(i)} - \alpha^{\tilde{z}_r(i-1)}\right)\left(\gamma - \tilde{z}_r^o(i-1)\right)\\
&\quad - \sum_{r:\exists\pi_i^o(\cdot)=r} \alpha^{\tilde{z}_r(i)}\tilde{w}_{r|\pi_i^o}(i),
\end{aligned}$$

where the last equality follows from the fact that $\alpha^{\tilde{z}_r(i)} - \alpha^{\tilde{z}_r(i-1)} = 0$ for all r that $\forall\pi_i(\cdot) \neq r$, and $\tilde{w}_{r|\pi_i^o}(i) = 0$ for all r that $\forall\pi_i^o(\cdot) \neq r$. Inequality (18) follows from $\tilde{z}_r^o(i-1) \ge 0$ and $\tilde{z}_r(i) = \tilde{z}_r(i-1) + \tilde{w}_{r|\pi_i}(i)$. Note that the first term in (18) is the same as the objective function (15). Because the mapping π_i results from Algorithm 2 which optimizes (15), we know that the reference mapping π_0 must produce a cost $\alpha^{\tilde{z}_r(i-1)+\tilde{w}_{r|\pi_i^o}(i)} - \alpha^{\tilde{z}_r(i-1)}$ that is greater than or equal to the optimum, hence following (19). Equality (20) is obvious.

Now we proof that the potential function $\Phi(i)$ does not increase with i, by proving that (20) is not larger than zero. For the ith request, the reference offline mapping produces the mapping result π_i^o. Therefore, for all r such that $\exists\pi_i^o(\cdot) = r$, we have $0 \le \tilde{w}_{r|\pi_i^o}(i) \le J_{\pi^o}/\hat{J} \le 1$. Hence, we only need to show that $\gamma\left(\alpha^{\tilde{w}_{r|\pi_i^o}(i)} - 1\right) - \tilde{w}_{r|\pi_i^o}(i) \le 0$ for $\tilde{w}_{r|\pi_i^o}(i) \in [0,1]$, which is true for $\alpha \le 1+1/\gamma$. From (17)–(20), it follows that $\Phi(i) \le \Phi(i-1)$. (We take $\alpha = 1+1/\gamma$ because this gives the smallest value of β.)

Because $\tilde{z}_r(0) = \tilde{z}_r^o(0) = 0$, we have $\Phi(0) = \gamma(NK+L)$. Because $\Phi(i)$ does not increase, $\alpha > 1$, and $\tilde{z}_r^o(i) \le 1$ due to $J_{\pi^o} \le \hat{J}$, we have

$$\begin{aligned}
(\gamma-1)\,\alpha^{\max_r \tilde{z}_r(i)} &\le (\gamma-1)\sum_r \alpha^{\tilde{z}_r(i)}\\
&\le \Phi(i)\\
&\le \Phi(0)\\
&= \gamma(NK+L). &(21)
\end{aligned}$$

Taking the logarithm on both sides of (21), we have

$$\max_r \tilde{z}_r(i) \le \log_\alpha\left(\frac{\gamma(NK+L)}{\gamma-1}\right) = \beta, \tag{22}$$

which proves the result because $z_r(i) = \tilde{z}_r(i)\cdot\hat{J}$.

ACKNOWLEDGMENT

The authors gratefully thank Dr. Moez Draief, Dr. Ting He, Dr. Viswanath Nagarajan, Dr. Theodoros Salonidis, and Dr. Ananthram Swami for their valuable suggestions to this work.

Contribution of S. Wang is partly related to his previous affiliation with Imperial College London. Contribution of M. Zafer is not related to his current employment with Nyansa Inc.

REFERENCES

[1] S. Wang, "Dynamic service placement in mobile micro-clouds," Ph.D. dissertation, Imperial College London, 2015.

[2] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[3] P. Bahl, R. Y. Han, L. E. Li, and M. Satyanarayanan, "Advancing the state of mobile cloud computing," in *Proceedings of the third ACM workshop on Mobile cloud computing and services*. ACM, 2012, pp. 21–28.

[4] Y. Abe, R. Geambasu, K. Joshi, H. A. Lagar-Cavilla, and M. Satyanarayanan, "vTube: efficient streaming of virtual appliances over last-mile networks," in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 16.

[5] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan, "Towards wearable cognitive assistance," in *Proc. of ACM MobiSys*, 2014.

[6] M. Satyanarayanan, R. Schuster, M. Ebling, G. Fettweis, H. Flinck, K. Joshi, and K. Sabnani, "An open ecosystem for mobile-cloud convergence," *IEEE Communications Magazine*, vol. 53, no. 3, pp. 63–70, Mar. 2015.

[7] "Smarter wireless networks," *IBM Whitepaper No. WSW14201USEN*, Feb. 2013. [Online]. Available: www.ibm.com/services/multimedia/Smarter_wireless_networks.pdf

[8] "Mobile-edge computing – introductory technical white paper," Sept. 2014. [Online]. Available: https://portal.etsi.org/tb.aspx?tbid=826&SubTB=826

[9] M. Satyanarayanan, G. Lewis, E. Morris, S. Simanta, J. Boleng, and K. Ha, "The role of cloudlets in hostile environments," *IEEE Pervasive Computing*, vol. 12, no. 4, pp. 40–49, Oct. 2013.

[10] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*. ACM, 2012, pp. 13–16.

[11] M. Peng, S. Yan, K. Zhang, and C. Wang, "Fog computing based radio access networks: Issues and challenges," *arXiv preprint arXiv:1506.04233*, 2015.

[12] T. Taleb and A. Ksentini, "An analytical model for follow me cloud," in *Proc. of IEEE GLOBECOM 2013*, 2013.

[13] Z. Becvar, J. Plachy, and P. Mach, "Path selection using handover in mobile networks with cloud-enabled small cells," in *Proc. of IEEE PIMRC 2014*, Sept. 2014.

[14] Y.-h. Chu, S. G. Rao, and H. Zhang, "A case for end system multicast (keynote address)," *SIGMETRICS Perform. Eval. Rev.*, vol. 28, no. 1, pp. 1–12, June 2000.

[15] B. Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Proc. of 22nd International Conference on Distributed Computing Systems Workshops*, 2002, pp. 575–578.

[16] D. Westhoff, J. Girao, and M. Acharya, "Concealed data aggregation for reverse multicast traffic in sensor networks: Encryption, key distribution, and routing adaptation," *IEEE Trans. on Mobile Computing*, vol. 5, no. 10, pp. 1417–1431, Oct. 2006.

[17] A. Ksentini, T. Taleb, and M. Chen, "A Markov decision process-based service migration procedure for follow me cloud," in *Proc. of IEEE ICC 2014*, Jun. 2014.

[18] S. Wang, R. Urgaonkar, M. Zafer, T. He, K. Chan, and K. K. Leung, "Dynamic service migration in mobile edge-clouds," in *Proc. of IFIP Networking 2015*, May 2015.

[19] 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, Sept. 2015.

[20] S. E. Mahmoodi, R. N. Uma, and K. P. Subbalakshmi, "Optimal joint scheduling and cloud offloading for mobile applications," *IEEE Transactions on Cloud Computing*, vol. PP, no. 99, pp. 1–1, 2016.

[21] W. Zhang, Y. Wen, and D. O. Wu, "Collaborative task execution in mobile cloud computing under a stochastic wireless channel," *IEEE Transactions on Wireless Communications*, vol. 14, no. 1, pp. 81–93, Jan. 2015.

[22] J. Liu, Y. Mao, J. Zhang, and K. B. Letaief, "Delay-optimal computation task scheduling for mobile-edge computing systems," in *2016 IEEE International Symposium on Information Theory (ISIT)*, Jul. 2016, pp. 1451–1455.

[23] A. Fischer, J. Botero, M. Beck, H. De Meer, and X. Hesselbach, "Virtual network embedding: A survey," *IEEE Commun. Surveys Tuts.*, vol. 15, no. 4, pp. 1888–1906, 2013.

[24] I. Giurgiu, C. Castillo, A. Tantawi, and M. Steinder, "Enabling efficient placement of virtual infrastructures in the cloud," in *Proceedings of the 13th International Middleware Conference*, ser. Middleware '12, 2012, pp. 332–353.

[25] V. V. Vazirani, *Approximation Algorithms*. Springer, 2001.

[26] A. Borodin and R. El-Yaniv, *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.

[27] M. Chowdhury, M. Rahman, and R. Boutaba, "Vineyard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 206–219, 2012.

[28] N. Bansal, K.-W. Lee, V. Nagarajan, and M. Zafer, "Minimum congestion mapping in a cloud," in *Proceedings of the 30th annual ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, ser. PODC '11, 2011, pp. 267–276.

[29] D. Dutta, M. Kapralov, I. Post, and R. Shinde, "Embedding paths into trees: VM placement to minimize congestion," in *Algorithms – ESA 2012*, ser. Lecture Notes in Computer Science, L. Epstein and P. Ferragina, Eds. Springer Berlin Heidelberg, 2012, vol. 7501, pp. 431–442.

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

[31] J.-J. Kuo, H.-H. Yang, and M.-J. Tsai, "Optimal approximation algorithm of virtual machine placement for data latency minimization in cloud systems," in *Proc. of IEEE INFOCOM 2014*, 2014.

[32] T. W. Kuo, B. H. Liou, K. C. J. Lin, and M. J. Tsai, "Deploying chains of virtual network functions: On the relation between link and server usage," in *Proc. of IEEE INFOCOM 2016*, Apr. 2016, pp. 1–9.

[33] M. Rost and S. Schmid, "Service chain and virtual network embeddings: Approximations using randomized rounding," *arXiv preprint arXiv:1604.02180*, 2016.

[34] T. Lukovszki and S. Schmid, "Online admission control and embedding of service chains," in *Structural Information and Communication Complexity: 22nd International Colloquium, SIROCCO 2015, Montserrat, Spain, July 14-16, 2015. Post-Proceedings*, 2015, pp. 104–118.

[35] R. Hassin, A. Levin, and M. Sviridenko, "Approximating the minimum quadratic assignment problems," *ACM Trans. Algorithms*, vol. 6, no. 1, pp. 18:1–18:10, Dec. 2009.

[36] A. R. Choudhury, S. Das, N. Garg, and A. Kumar, "Rejecting jobs to minimize load and maximum flow-time," in *Proc. of ACM-SIAM Symposium on Discrete Algorithms (SODA)*, Jan. 2015.

[37] Y. Azar, "On-line load balancing," *Theoretical Computer Science*, pp. 218–225, 1992.

[38] W. B. Powell, *Approximate Dynamic Programming: Solving the curses of dimensionality*. John Wiley & Sons, 2007.

[39] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts, "On-line routing of virtual circuits with applications to load balancing and machine scheduling," *J. ACM*, vol. 44, no. 3, pp. 486–504, May 1997.

[40] IBM CPLEX Optimizer. [Online]. Available: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/

[41] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. R. Pruhs, and O. Waarts, "On-line load balancing of temporary tasks," *Journal of Algorithms*, vol. 22, no. 1, pp. 93–110, 1997.