IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

SPECIAL SECTION ON COLLABORATION FOR INTERNET OF THINGS

Received August 18, 2018, accepted September 12, 2018, date of publication September 20, 2018, date of current version October 17, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2871475

Composition-Driven IoT Service Provisioning in Distributed Edges

SHUIGUANG DENG[1], (Senior Member, IEEE), ZHENGZHE XIANG[1], (Student Member, IEEE), JIANWEI YIN[1], JAVID TAHERI[2], (Member, IEEE), AND ALBERT Y. ZOMAYA[3], (Fellow, IEEE)

[1]College of Computer Science and Technology, Zhejiang University, Hangzhou 310058, China
[2]Department of Mathematics and Computer Science, Karlstad University, SE-651 88 Varmland, Sweden
[3]School of Information Technologies, The University of Sydney, Sydney, NSW 2006, Australia

Corresponding author: Shuiguang Deng (dengsg@zju.edu.cn)

This work was supported in part by the National Key Research and Development Program of China under Grant 2017YFB1400601, in part by the Key Research and Development Project of Zhejiang Province under Grant 2015C01027 and Grant 2017C01015, in part by the National Science Foundation of China under Grant 61772461, and in part by the Natural Science Foundation of Zhejiang Province under Grant LR18F020003 and Grant LY17F020014.

ABSTRACT The increasing number of Internet of Thing (IoT) devices and services makes it convenient for people to sense the real world and makes optimal decisions or complete complex tasks with them. However, the latency brought by unstable wireless networks and computation failures caused by constrained resources limit the development of IoT. A popular approach to solve this problem is to establish an IoT service provision system based on a mobile edge computing (MEC) model. In the MEC model, plenty of edge servers are placed with access points via wireless networks. With the help of cached services on edge servers, the latency can be reduced, and the computation can be offloaded. The cache services must be carefully selected so that many requests can by satisfied without overloading resources in edge servers. This paper proposes an optimized service cache policy by taking advantage of the composability of services to improve the performance of service provision systems. We conduct a series of experiments to evaluate the performance of our approach. The result shows that our approach can improve the average response time of these IoT services.

INDEX TERMS Mobile edge computing, Internet of Thing, service provisioning, service composition.

I. INTRODUCTION

We are now embracing an era of IoT. With the help of radio frequency identification devices (RFID), near field communication (NFC), sensors, actuators mobile phones, etc., IoT technology empowers information systems to see, hear, think and perform jobs with the data collected from real world [1]. According to the report from Cisco, the IoT will consist of 50 billion devices connected to the Internet by 2020 [2]. There will be no doubt that IoT will play a more and more important role and will remold the communication between people and machines.

However, the instability of wireless communication and limited resource of IoT devices prevent users from experiencing high efficiency and seamless user experience. For example, the low computational capability and energy storage [3]–[5] of wireless devices restrict the popularization of novel services such like AR (Augmented Reality)/ VR (Virtual Reality)/ AI (Artificial Intelligence), and the packet losses cause external waiting time for urgent messages. Mobile Edge Computing (MEC) technology is proposed to solve some relevant problems for the aforementioned services [6]. MEC is a novel model that emerges recently as a reinforcement of mobile cloud computing to optimize the resource usage of IoT devices and wireless network to provide context-aware services. A typical architecture of MEC system is shown in Fig.1, where computation and transmission between devices and cloud servers are partly migrated to mobile edge servers.

To accelerate the services with the short-distance connections and the computation capacity of edge servers, a direct way is to use cache on these servers. However, traditional researchers pay most of their attention to the cache policy on data and ignore the scenarios of service cache, let alone taking advantages of service properties to optimize the cache policies. To take a step forward, we consider the scenarios of service cache on edge servers in the MEC architecture.

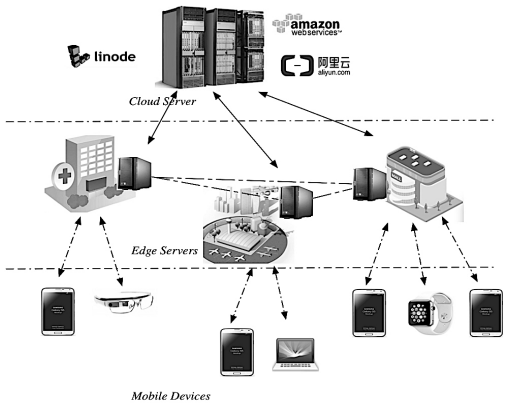

FIGURE 1. Architecture of mobile edge computing.

In this situation, mobile edge servers maintain different sets of services of the cloud server. Therefore, when the desired services are found on some edge servers, wireless devices can communicate with these mobile edge servers in the same way as with the cloud server if the service interfaces are unified. Hence, the effectiveness of a service provision system is strongly correlated to the cached service set of edge servers: the more popular services the mobile edge server cache have, the more requests from wireless devices can be optimized. It's also worth noting that, although for some services (such as video services) the communication with the cloud itself is still required, having servers on the edge of the network still plays a major role in improving the experience of users. For example, if the YouTube service is placed on the edge, then videos can be cashes/replayed on the edge by edge servers with powerful links to cloud servers. As a result, mobile users not only use their precious/limited mobile 3G/4G/LTE bandwidths to access videos on the cloud, but also get a better access to them as edge servers are presumed connected via fixed links with higher bandwidth and lower latency (than all types of mobile wireless connectivity). Therefore, in this work, we only consider the cost of bringing services to the edge, and assume the cost of transmitting data between edge and cloud servers negligible. What's more, with the developing of Micro-Service technology [7], services will not only work individually but also will cooperate with others easily to make a composite service and finish complex tasks. Thus, some composite services may also have the capability to fulfill some simple tasks by reusing their member services. In many cases, it will help save resource if both the composite services and their member services are invoked frequently. To take all things into consideration, in this paper we evaluate the performance with the average service response time (ASRT) and propose a resource consumption aware algorithm to determine how services can be placed on edge servers.

The contribution of this paper are listed as follows:

- In this paper, we explore the architecture of mobile edge computing benefit of service cache, and discuss the factors that may affect the performance in latency reduction of cache polices.
- We classify the services as composite services and atomic services, and use service composition graph to describe the relation of them. By considering the resource consumption, popularity (or invocation frequency) and the service composition graph, we propose an heuristic algorithm to optimize the average service response time.
- We conduct a series of experiments to evaluate the generated cache polices in algorithm and show the improvement.

The rest of this paper is organized as follows. Section 2 describes the motivation and scenario on cache policy with service composition. Section 3 highlights the related work of edge computing and the corresponding approaches. Section 4 presents definitions, concepts and components of the proposed problem. Section 5 describes the approaches we proposed to solve this problem. Section 6 shows the experimental results including the factors that affect our algorithms. Finally, Section 7 concludes our contribution and outlines future work.

II. MOTIVATION AND SCENARIO

In MEC architecture, the efficiency and effectiveness of service deployment/cache can strongly affect the performance of the system. At the same time, because edge servers are resource-constrained machines that can only cache finite number of services, resource consumptions of services must be considered during their selection.

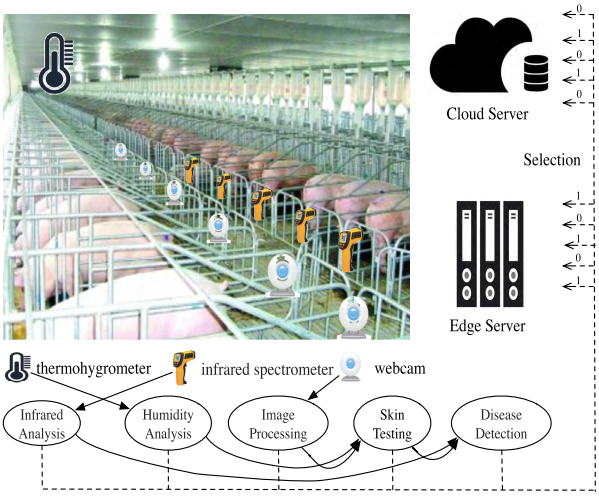

FIGURE 2. An example of pig farming industry.

To give an example of how service cache policies can affect the performance of provision system, we are going to consider a practice in modern pig-farming industry. Nowadays, with the help of IoT technology, the environment parameters can be collected in time so that farmers can easily monitor the pigpens — the thermohygrometer will record the humidity of the pigpens, the infrared spectrometer will scan the bodies of pigs and the webcam will record the behaviors of pigs, as shown in Fig.2. It is hard to capture the key information with the raw data, so researchers have developed serval

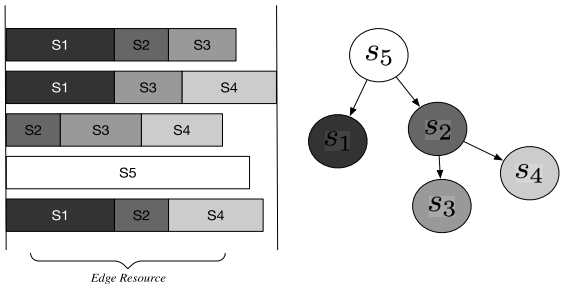

FIGURE 3. An example of edge server cache.

TABLE 1. Running parameters of services.

Service	t_i(s)	T_i(s)	Memory(MB)	Frequency
s_1	1.0	3.0	40	5%
s_2	2.5	4.0	30	15%
s_3	1.0	2.0	25	12%
s_4	2.0	4.0	35	23%
s_5	4.0	5.5	90	45%

services to analyze the data collected from such sensors. For example, there are 5 services — *Infrared Analysis* (s_1), *Humidity Analysis* (s_3), *Image Processing* (s_4), *Skin Testing* (s_2), and *Disease Detection* (s_5) developed by researchers to help understand the context of the pigpen. The service *Infrared Analysis* receives the real time data from infrared spectrometer and generate temperature distribution and stable heat maps of objects; The service *Humidity Analysis* uses the humidity sequence to generate the statistics charts and trends; The service *Image Processing* detects the objects of real time video stream and labels them different tags; The service *Skin Testing* check the skin quality of pigs to validate whether they need washing; The service *Disease Detection* uses some classification model to judge whether a pig is healthy or not. These 5 services can work individually to help decide whether it is necessary to ventilate the room, open the air conditioner, alarm the exceptional behaviors, wash the pigs or call for veterinarians. But actually, they have logical relations between each others. The service *Disease Detection* can use the results of service *Infrared Analysis* and service *Skin Testing* to make decisions, because the model works well with the heal map and skin quality as features. And the service *Skin Testing* can take advantages of humidity and the features from image processing to judge whether the skin is normal. Therefore, some complex tasks can be fulfilled by invoking some of the simple services. The composition relations of these services are shown in Fig.3 in which the edge $s_m \rightarrow s_n$ means that s_n is a component of service s_m. It models realistic scenarios where different Docker images (e.g. s_m) may include the same container (e.g. s_n), and thus if the component containers can be reused by modifying the parameters, the resource can be reused. Besides these, the properties of these services are shown in Table.1. These services are to be selected for cache on an edge server (es) whose maximum available memory is 100MB. In Table.1, t_i and T_i means the response time of services when it is deployed on the edge (i.e. cached) and on the cloud, respectively.

In this situation, we need to determine which services can be cached in edge server, so that it can swiftly deal with sensors' requests (in this paper, we use the ASRT to evaluate the performance). A direct idea is to store the popular services because they are invoked frequently, if these popular services are accelerated with cache, the average response time is expected to be reduced. However, in this example (Fig.3), the most popular service (s_5) will use nearly all the computing resource of the edge server, and therefore when it is cached on the edge server, the ASRT will be $\sum_{i=1}^{4} T_i \times f_i + t_5 \times f_5 = 3.71$s, which is greater than the average response time (3.58s) of selecting less popular services (s_2, s_3, s_4). Using this simple example, we showed that it is necessary to build more complex models to make such caching decisions. Please also note that service s_5 is a composite service made up of service s_1 and s_2, where s_2 is made up of s_3 and s_4. These composition relations are quite common in practice. Here for example, we can take advantage of such information and cache services (s_1, s_3, s_4) to achieve the ASRT of $t_1 \times f_1 + (t_3 + t_4) \times f_2 + t_3 \times f_3 + t_4 \times f_4 + (t_1 + t_3 + 4_4) \times f_5 = 2.88$s, which is less than the former ones.

III. RELATED WORK

With an increasing number of wireless devices connecting to the cloud, the demand for high-quality service provision becomes urgent. It drives more and more researchers to pay attention to issues of the MEC model that affect the effectiveness of service provision. In this section, we review the research related to our study, i.e. MEC framework and cache policy.

A. IoT AND MEC FRAMEWORK

With the help of MEC model, researchers and developers reconstruct their system components to achieve their different goals. Since the MEC model focuses on the mobile end devices, energy consumption reduction and performance optimization become the main research topics to perform computation in an economical and efficient way. For example, Tianze *et al.* [8] consider the energy consumption of wireless devices; they analyze overheads of wireless devices and propose an overhead-optimizing multi-device task scheduling strategy for ad-hoc-based MEC systems. Sardellitti *et al.* [9] consider a Multi-Input and Multi-Output (MIMO) multi-cell system where multiple mobile users ask for computation offloading to servers; they formulate the offloading problem as the joint optimization of the radio resources and the computational resources to satisfy the latency constraints. You *et al.* [10] consider incorporation with dense cellular networks; they propose an online algorithm based on Lyapunov optimization to determine offloading and edge server sleeping policy and increase performance while keeping low energy consumption. Different from traditional QoS performance prediction optimization, Wang *et al.* [11] proposed an efficient QoS prediction

approach for service recommendation that considers user mobility and data volatility for the first time. Yi *et al.* [12] propose LAVEA, a system built for edge computing, which offloads computation tasks between clients and edge nodes, collaborates nearby edge nodes, to provide low-latency video analytics at places closer to the users.

B. CACHE POLICY

In computer architectures, caches are among the most popular approaches to help improving system performance. Researchers tries to determine what/where/how to place their data and services of specific systems with different properties. For example, Bahreini and Grosu [13] focus on the placement of the components of a mobile application on the edge servers that minimizes the cost of running the application. What's more, they also consider the impact of users' location and network capacity in their optimization problem. Gu *et al.* [14] investigate the storage allocation of macro base station. Bai *et al.* [15] has proposed a caching based device-to-device communication scheme. They consider the social relations among users and their common interests. Breslau *et al.* [16] suggests to use the zipf-like model to describe the content popularity in web caching. Ahlehagh and Dey [17] proposes a user preference profile based caching policy. They find that video popularity varies in different users and propose a caching policy on these user preferences.

These research shed light on the fundamental concepts of the cache problem in MEC models, but they do not cover the cache policy on the mobile edge servers by utilizing the composition technology of services.

IV. PROBLEM DEFINITION AND DESCRIPTION

In this section, we introduce the related definitions and descriptions involved in the cache content optimization problem of MEC systems.

A. CONCEPT DEFINITION

Definition 1 (Cloud Server, CS): Cloud Server consists of clusters of machines. It maintains a service registry and acts as a service provision center. Service providers can register, delegate and maintain their services on it, while user (or sensor) can query or invoke the services from it. We define $V_{u\rightarrow cs}$ as the average transmission rate between cloud server and wireless users.

Though the cloud server is made up of machines, some management systems like OpenStack, Kubernetes[1] are applied in practice to manage these machines. Hence, the cloud server can run in a well-organized way and provide unified API to end users.

Definition 2 (Web Service, WS): A Web Service *ws* can be defined by $ws = (func, in, out, r, QoS)$, where *func* is the functionality of the web service, *in* is the input, *out* is the output, r is the resource consumption, and *QoS* is the quality of web service.

[1]https://kubernetes.io

The web service is an abstract concept that describes how a program can complete some tasks with specific parameters. It can be an instance of remote invocation based on SOAP [18], a web API provided by a software company, or a Docker[2] container/ Kubernetes pod managed by a PaaS platform. In our work, we focus on the Container-as-a-Service situation because of container's "*Build, Ship, and Run Any App, Anywhere*" properties. Edge servers can download application images from clouds and run containers to deal with different requests.

```
apiVersion: v1
kind: ReplicationController
metadata:
    name: nginx-controller
spec:
    containers:
    - image: nginx
      name: nginx
      resources:
        limits:
          memory: 200Mi
        requests:
          memory: 100Mi
```

FIGURE 4. An example of Kubernetes pod YAML file.

The r describes resource consumption of services such as memory, CPU, GPU or network. For example, Fig. 4 shows a description of the nginx-pod. *QoS* describes the nonfunctional characteristics of service, including execution time, reputation, cost, etc. [19], [20]. It varies when deployed in different environments. In our model, we focus on execution time of web services on different edge and cloud servers.

Definition 3 (Service Composition Graph, SCG): A Service Composition Graph reveals the relations of web services, it can be represented as a directed acyclic graph (DAG) $G_{sc} = (S, E)$, where S is the web service set, and $E = \{< s_i \leftarrow s_j > | s_i, s_j \in S, (s_i, s_j) \in R_c\}$ is the set of graph edges. Here, R_c is the composition structure of services, in which $(s_i, s_j) \in R_c$ means that service s_j is a composite service and s_i is a member service of s_j.

The service composition technology is derived from service-oriented architecture (SOA), it tries to compose new complex services by integrating services with various functionalities provided by different service developer [21], [22]. For example, Siri[3] is one of the most famous composite applications, it integrates member services like alarm clock, calendar, weather, music, etc. by recognizing human voice and responses to users by dialogs. There exist many other composite services in the service registry.[4] With well-designed UI wrappers and runtime optimization (like allocating different CPU units for parallel service execution), these composite services sometimes show better user experience and performance than invoking individual member services [23].

[2]https://www.docker.com
[3]https://www.apple.com/cn/ios/siri/
[4]https://www.programmableweb.com/category/all/mashups

In the graph, the services which are made of other member services are called "*composite service*" (S_c), and the others are called "*atomic service*" (S_a). If a service s_i is a member service of service s_j, we denote that service s_i *support* service s_j.

Definition 4 (Edge Server, ES): Edge servers are computing devices with stronger processing power and larger storage capacity that deployed on the edge network. The edge server communicates with wireless devices (usually via wireless links), receives their requests and records the running events. An edge server es can be defined by a tuple $es = (R, V_{u\to es}, V_{cs\to es})$, where R is the provision capacity, $V_{u\to es}$ is the data transmission rate between users and edge server, and $V_{cs\to es}$ is the data transmission rate between cloud servers and edge servers.

In our model, we use average transmission rate $V_{u\to es}$ and $V_{cs\to es}$ in response time computation to reduce the computation complexity. The parameter $R = (R^{res_1}_{max}, R^{res_2}_{max}, \ldots, R^{res_n}_{max})$ describes the maximum resource that the edge server can provide in different resource types. For example, an MEC provision system which considers only the memory resource will give the edge server a provision capacity; e.g., $R = R^{mem}_{max}$. Every service will use resources of different types, but the sum of each individual resource cannot be larger than the capacity of the edge servers.

B. PROBLEM DESCRIPTION

A significant metric to measure the performance of a service provision system is the average service response time. Short service response time will significantly enhance user experience and can motivate users to use more services. For every web service ws_i, it typically has two different service response times T^c_i and t^e_i. T^c_i is the service response time when it is placed on the cloud server, and t^e_i is on the edge server. They can be easily derived from the parameters of edge server and service:

$$\begin{cases} t^e_i = \dfrac{in_i}{V_{u\to es}} + QoS^{edge}_{exec\_time} + \dfrac{out_i}{V_{u\to es}} \\ T^c_i = \dfrac{in_i}{V_{u\to cs}} + QoS^{cloud}_{exec\_time} + \dfrac{out_i}{V_{u\to cs}} \end{cases} \tag{1}$$

In many cases, it is assumed that tasks will execute better on cloud because the machines of the cloud may have better hardware for computation. But things are not absolute because edge servers are also machines with good computation capability, while the cloud is made up of such machines. Therefore, we do not assume whether it takes longer on edge server or on cloud server. But in general, the transmission time when services deployed on edge servers is always larger than that on cloud, because of the short-distance communication with edge and shorter waiting time in the serving queue.

Besides invoking services from cache or cloud directly, the task of a service can also be fulfilled by invoking its member services. If service ws_i is a composite service that consists of other services $MS_i = \{s_{i1}, s_{i2}, \ldots, s_{ik}\}$, the service response time $t^*_i(y)$ can be computed recursively with the service response time of the member services (here y is the vector that describe the cache policy where $y_i = 1$ means $s_i \in$ cache and $y_i = 0$ means $s_i \notin$ cache):

$$t^*_i(y) = \begin{cases} \min\{t^e_i, \sum_{s_j\in MS_i} t^*_j\}, & y_i = 1, s_i \in S_c \\ \min\{T^c_i, \sum_{s_j\in MS_i} t^*_j\}, & y_i = 0, s_i \in S_c \\ t^e_i, & y_i = 1, s_i \notin S_c \\ T^c_i, & y_i = 0, s_i \notin S_c \end{cases} \tag{2}$$

Then, the ASRT $\widetilde{rt}(y)$ of S can be represented as:

$$\widetilde{rt}(y) = f_1 \times t^*_1(y) + f_2 \times t^*_2(y) + \ldots + f_n \times t^*_n(y) \tag{3}$$

$$= \sum_{i=1}^{n} f_i \times t^*_i(y) \tag{4}$$

Here f_i is the frequency of the service s_i to reflect its popularity. The frequency can be estimated by counting the invocations of different services.

TABLE 2. Symbols and notations.

Symbol	Meaning
cs	The cloud server
es	The edge server
$V_{u\to cs}$	The average data transmission rate between device and cloud server
$V_{cs\to es}$	The average data transmission rate between cloud server and edge server
$V_{u\to es}$	The average data transmission rate between device and edge server
s_i	A service that fulfills a specific task
S	The service set
G_{sc}	The service composition graph
S_c	The set of composite service
S_a	The set of atomic service
r_i	The resource consumption of service s_i
R	The resource limitation of edge server
f_i	The popularity (frequency) of service s_i
t^e_i	The service response time when s_i placed on es
T^c_i	The service response time when s_i placed on cs
MS_i	The set of member service of s_i
y_i	The selection indicator for s_i

In this way, with the notations and symbols shown in Table. 2, the problem can be defined as: For an arbitrary edge server $es = (R, V_{u\to es}, V_{cs\to es})$, given the services $S = \{s_1, s_2, \ldots s_n\}$, the corresponding service composition graph $G_{sc} = (S, E)$ and the popularity $f = \{f_i\}^n_{i=1}$ of services, finding the optimal service cache policy $y = \{y_i\}^n_{i=1}$ for edge server es to minimize the average service response time $\widetilde{rt}(y)$.

$$\min\ \widetilde{rt}(y) = \sum_{i=1}^{m} f_i t^*_i(y) \tag{5}$$

$$s.t.\ \begin{cases} \sum_{i=1}^{m} y_i r_i \le R \\ y_i \in \{0, 1\} \end{cases} \tag{6}$$

V. APPROACH

In this section, we will firstly analyze the relation of services based on the service composition graph, and then propose a heuristic algorithm based on this analysis.

Algorithm 1 Computing Service Response Time, CSRT

Input:
$SCG(S, E)$: the service composition graph;
$T = \{T_i\}_{i=1}^{n}$: the response time on cloud;
$t = \{t_i\}_{i=1}^{n}$: the response time on edge if cached;
y: the cache policy;
Output:
$\{t_i^*\}$: response time with cache policy y;

1 *Stack* $S_\dagger \leftarrow \varnothing$
2 **while** *not all computed* **do**
3 $S_{zero} \leftarrow \{s_i | s_i \in S, indegree(s_i) = 0\}$
4 **for** $s_k \in S_{zero}$ **do**
5 **if** $outdegree(s_k) = 0$ **then**
6 **if** $y_k = 1$ **then**
7 $t_k^* = t_k^e$
8 **else**
9 $t_k^* = T_k^c$
10 **else**
11 $push(S_\dagger, s_k)$
12 **for** $s_k \in S_{zero}$ **do**
13 $remove(G, s_k)$
14 *update* G
15 **while** $S_\dagger \neq \varnothing$ **do**
16 $S_k \leftarrow pop(S_\dagger)$
17 **for** $s_k \in S_k$ **do**
18 **if** $y_k = 1$ **then**
19 $t_k^* = \min\{\sum_{s_j \in MS_k} t_j^*, t_k^e\}$
20 **else**
21 $t_k^* = \min\{\sum_{s_j \in MS_k} t_j^*, T_k^c\}$
22 **return** $\{t_1^*, t_2^*, \ldots, t_n^*\}$

A. SERVICE COMPOSITION GRAPH ANALYSIS

The service composition graph reveals the hierarchal structure of web services. With this information, appropriate services can be selected to support more service requests using edge server caches.

To represent the service response time of all services with an analytical expression, we proposed an iterative algorithm that takes advantage of the *support* relations among services; it is presented in algorithm 1. In Algorithm 1, it firstly partitions services to different levels and uses a stack to store them: the higher level a service has, the lower it will stay in the stack (Line 2-14). Then the response time are computed level by level (Line 15-21). In the worst case, there is only one path that connects all services like $s_1 \leftarrow s_2 \leftarrow s_3$, in this situation the time complexity is $O(|S|^2)$.

To illustrate the computation process, we show an example of how the response time is computed with a given policy in Fig.5. In this case, there are 15 services in the service registry which are numbered from s_0 to s_{14}. In step (1), the services whose indegree is 0 (s_0, s_1, s_6, s_7, s_9, s_{13}, s_{14}) are highlighted. In step (2), the response times of services whose outdegree is 0 (atomic services) are computed directly with the value of y, and then these services are removed from the service composition graph. In step (3), the services whose outdegree is larger than 0 (composite services) are stored in the stack and these services and their related edges are also removed from the graph. Step (4) and (5) repeat the process from (1) to (3). In step (6), all the composite services are stored in stack in their level order. In step (7), the response time of service s_2 is computed because all its member services' response time ($t_3^*(y)$, $t_5^*(y)$) are computed. When $t_2^*(y)$ is worked out, the stack will pop it. In step (8), the response times of s_1 and s_{13} are worked out, and because the stack is now empty, the computing process is over.

B. CACHE POLICY ALGORITHMS

Inspired by the traditional knapsack problem [24], the main step of making cache policy is to decide which service is appropriate to put in cache so that the ASRT can be minimized. Furthermore, the value of policy is however not as easy to evaluate as that in the knapsack problem, because the relations here are much more complex.

1) ENUMERATION METHOD

A direct way is to enumerate all possible policies to find the optimal one. Enumeration is a brute-force but accurate method to find the optimal solution of this NP-complete problem. As the size of the solution set is 2^n, this approach enumerates all polices from $y = <0, 0, \ldots, 0>$ to $<1, 1, \ldots, 1>$, compute and compare the ASRT in turn, the policy with minimum ASRT is the optimal one. We choose the results of it as the ground truth of our experiments.

2) CONSUMPTION-DRIVEN SEARCHING ALGORITHM

Genetic algorithm (GA) is a kind of metaheuristic inspired by the process of natural selection. It simulates the evolution of populations with operations like *selection*, *crossover* and *mutation*. GA is designed to favor chromosomes with highest fitness values to produce next populations (solutions). As a result, quality of solutions for a problem is gradually improved (population by population) until the optimal answer is reached. Inspired by GA, we propose the consumption-driven searching algorithm (CDSA) with the following three steps:

(1) Encoding. Encoding is the first step of the CDSA algorithm. In this step, the solutions of the optimization problem are represented with encoded chromosome firstly. From the problem description, the goal is to find optimal cache policy $\vec{y} = <y_1, y_2, \ldots, y_n>$ which can minimize the ASRT. We encode the candidate policies with an n-bits-genome chromosome where 0 or 1 in the i-th genome means the selection of service ws_i.

Secondly, a set of chromosomes are initialized to make a population. Typically, the several hundreds or thousands of possible solutions are contained in a population, and the

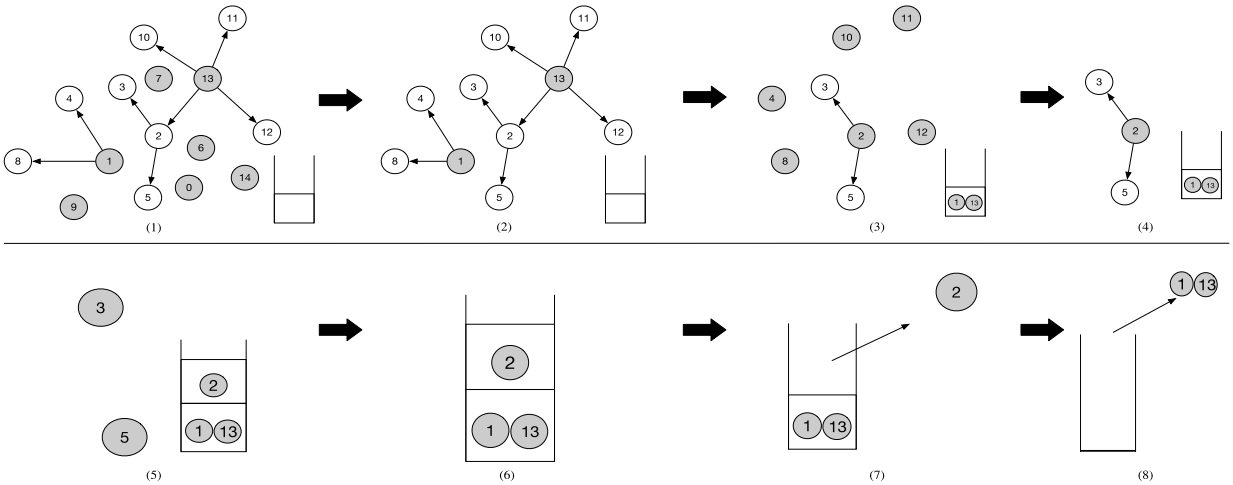

FIGURE 5. An example of Algorithm 1.

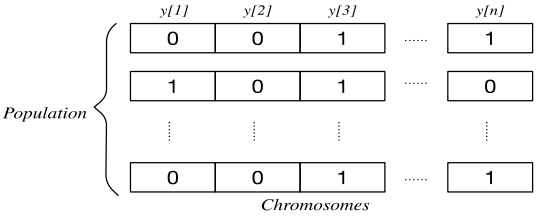

FIGURE 6. Genomes and populations in a generic algorithm.

chromosomes of the population are generated randomly to cover a wide solutions space in increase the chance of finding the optimal solution. However, because the overuse of resources is forbidden in the provision system, random initialization may result in unrealistic solutions. Furthermore, the entire searching space will have 2^n points if it is initialized randomly without constraints, which will make it difficult for the algorithm to find the optimal solutions. Thus, in our approach, we divide the population into 2 parts. The chromosomes of the first part are initialized randomly to keep the algorithm able to escape local optimums; in the second part, we initialize the chromosomes with the resource consumption constraint according to algorithm 2. We will prove that the generative model G can generate all valid policies that satisfies (6).

Lemma 1: Denote the policies generated by Algorithm 2 with P_g and the valid policies with P^*, then $P_g = P^*$.

Proof: To prove the equality of P_g and P^*, we only need to prove that 1) $\forall p \in P_g, p \in P^*$ and 2) $\forall p \in P^*, p \in P_g$.

1) Given an arbitrary $p \in P_G$, $p = \{y_1, y_2, \ldots, y_n\}$, the assignment order of y_k can be described by collecting the results of *pop* operation in line 9 of Algorithm 2 as $O_p = \{y_{o_1}, y_{o_2} \ldots, y_{o_n}\}$, then before the y_{o_i} is assigned with 0 or 1, the policy with assigned $\{y_{o_1}, y_{o_2} \ldots, y_{o_{i-1}}\}$ are ensured to be valid, because the available resource R in Algorithm 2 is still non-negative; It means that there are still resource to be allocated for the remaining services. For service s_{o_i}, if it is selected to be cached on edge server, the remaining resource will be $R = R - r_{o_i}$. If $R \leq 0$, the allocation will be withdrawn in Line 5 of Algorithm 2. Finally, the left resource R is still non-negative, and the policy p is generated. In this way, we get $p \in P^*$.

2) Given an arbitrary $p \in P^*$, we can replay the generating process by replacing the *RandomPop* in Line 9 of Algorithm. 2 by selecting specific services. Because p is the valid policy, the remaining resource R must by non-negative after adopting p, so it is valid for every step in the generation. In this way, due to feasibility in replaying the generating process in Algorithm 2, we have $p \in P_g$.

Consequently, we can prove that $P_g = P^*$. [Q.E.D]

The initialization algorithm has the runtime complexity of $O(|S|)$ and can be easily parallelized.

(2) Selection. During each successive generation, a portion of the existing population is selected to breed a

Algorithm 2 Initialization for CDSA

Input:
$\{r_i\}_n$: the resource consumptions of services;
R: the resource capacity of target edge server;
Output:
$y = \{y_1, y_2, \ldots, y_n\}$: the cache policy of services;

```
1  S ← {s_1, s_2, ..., s_n}
2  A ← ∅
3  while S ≠ ∅ do
4      if R ≤ 0 then
5          (s_w, y_w) ← Pop(A)
6          y_w ← 1 − y_w
7          Push(A, (s_w, y_w))
8      s_k ← RandomPop(S)
9      y_k ← Random{0, 1}
10     if y_k = 1 then
11         R ← R − r_k
12     Push(A, (s_k, y_k))
13 return y
```

new generation. Individual solutions are selected through a fitness-based process, where fitter solutions are typically more likely to be selected. The fitness function of GA measures the quality of generated solution. In our approach, as the objective is to minimize the ASRT with the constraint on resource consumption, the fitness function F can be defined by:

$$F(y) = \begin{cases} 1/\sum_{i=1}^{n} f_i \cdot t_i^*(y), & R \geq \sum_{i=1}^{n} r_i \cdot y_i \\ 0, & otherwise \end{cases} \quad (7)$$

In this way, a solution with smaller ASRT will have larger probability to be selected. By computing all the fitness value of chromosomes of the population, the chromosomes $y^{(k)}$ are chosen according to their probability by revolving a roulette in which the k-th part occupies $P^{(k)}$ percentage of it. Here, $P^{(k)}$ is the probability to select $y^{(k)}$ to produce new chromosomes for the next generation (the selecting process is liking rotating a roulette shown in Fig. 7).

$$P^{(k)} = \frac{F(y^{(k)})}{\sum_{i=1}^{n} F(y^{(i)})} \quad (8)$$

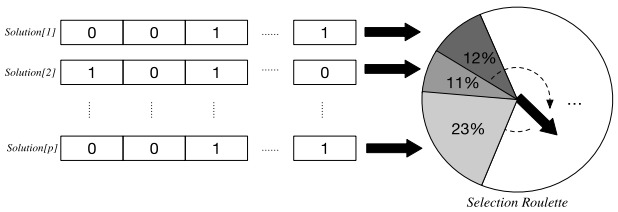

FIGURE 7. Selecting chromosomes with a roulette.

(3) Crossover and Mutation. With chromosomes prepared in a population, the following step is to generate the next generation of solutions. For each new solution to be produced, a pair of "parent" genomes is selected with possibility reflected in (8). Firstly, a new solution is created by sharing many of the characteristics of these "parents". It means that the selected "parents" exchange parts of their bits with each other. On the other hand, the selected "parents" may choose not to crossover, then the new "offspring" are identical to themselves. We use the parameter *crossover probability* (p_c) to determine how new chromosomes are produced. The process continues until a new population of solutions of appropriate size is generated. Secondly, mutations may occur on the newborn populations. In the mutation, some genomes of a chromosome may change with a low possibility p_m called *mutation probability*. The mutation operation gives the algorithm the ability to avoid premature convergence. At last, several solutions with good fitness will stay unchanged as elites in next generation to keep the convergence. This process finally stops when converged after 5 consecutive iterations, it results in solutions with appropriate fitness values. Choosing the one with best fitness value from the final population, the corresponding policy will be the suboptimal of the problem.

The algorithm. 3 shows the process. Firstly, the solution set or population is initialized with algorithm. 2 (Line 2–7).

Algorithm 3 Consumption-Driven Searching Algorithm, CDSA

Input:
PopSize: the size of population;
$rand_{\%}$: the percentage for random initilization;
MaxEpoch: the maximum epoch number for evolution;
Output:
$y = \{y_1, y_2, \ldots, y_n\}$: the cache policy of services;

1 $Population_{old} \leftarrow \varnothing$
2 **while** $|Population_{old}| < PopSize$ **do**
3 **if** $|Population_{old}| \leq PopSize \times rand_{\%}$ **then**
4 $chromosome \leftarrow RandInit()$
5 **else**
6 $chromosome \leftarrow CDSAInit()$
7 $Add(Population_{old}, chromosome)$
8 $fitness \leftarrow Evaluate(Population_{old})$
9 $T \leftarrow 0$
10 **while** $!convergence\ or\ T < MaxEpoch$ **do**
11 $T \leftarrow T + 1$
12 $Population_{new} \leftarrow \varnothing$
13 $Elitism \leftarrow SelectElitism(Population_{old}, E_{num})$
14 $Add(Population_{new}, Elitism)$
15 **while** $|Population_{new}| < PopSize$ **do**
16 $father \leftarrow Select(Population_{old})$
17 $mother \leftarrow Select(Population_{old})$
18 $child_1 \leftarrow Crossover(father, mother, p_c)$
19 $child_2 \leftarrow Crossover(father, mother, p_c)$
20 $child_1 \leftarrow Mutate(child_1, p_m)$
21 $child_2 \leftarrow Mutate(chile_2, p_m)$
22 $Add(Population_{new}, child_1)$
23 $Add(Population_{new}, child_2)$
24 $Population_{old} \leftarrow Population_{new}$
25 $fitness \leftarrow Evaluate(Population_{old})$
26 $policy \leftarrow SelectElitism(Population_{old}, 1)$
27 **return** *policy*

Secondly, The fitness of solutions or chromosomes are calculated with the *Evaluate*() function described in (7) (Line 8). Then for every generation, the algorithm will keep the elitism with good fitness and select parent chromosomes according to the probability in (8) and crossover them to generate offspring. The offspring may mutate in every generation (Line 10–25). The algorithm stops when the solution converges or the generation number exceeds *MaxEpoch*.

VI. EXPERIMENTS AND ANALYSIS

We have implemented the deploying algorithms in Python 2.7.13 and our experiments are conducted on a machine with Intel Xeon E5-2620 v4@2.10GHz × 2 CPU and 64GB memory on CentOS 7 operation system. Due to the lack of well adopted platforms and datasets, we generated

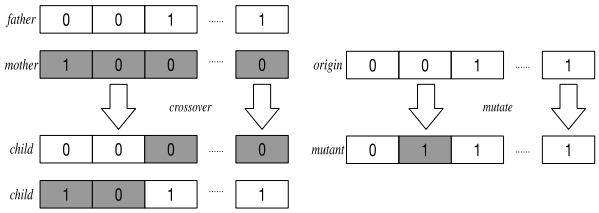

FIGURE 8. Crossover and mutation in every epoch.

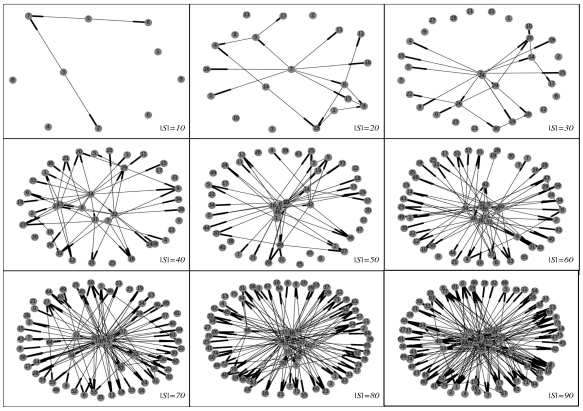

FIGURE 9. Some service composition graphs in dataset.

TABLE 3. Statistics of service composition graphs.

statistics	*min*	*max*	*mean*
$\|S\|$	5	100	52.5
η	0	0.67	0.29
$\|S_c\|_{\%}$	0%	86.67%	32.21%
H_s	0	0.82	0.43

our experimental data in a synthetic way. In our experiment, the most important data is the service composition graph to reveal the relation among services. Fig.9 shows some examples of service composition graphs generated to verify our approach with different service sets. There are atomic services and composite services in every service composition graph. Different composite services can include the same member service, and every two services in the same path of the graph will not include each other. Table. 3 shows the statistics information of the experiment data; here $|S|$ is the number of involved services, η is defined by $\frac{2|E|}{|S|(|S|-1)}$ which describes the complexity of the graph, $|S_c|_{\%} = \frac{|S_c|}{|S|}$ is the percentage of composite service, and $H_s = \frac{\max_{p \in path} |p|}{|S|}$ describes how complex the hierarchy of the composite can be.

A. EVALUATION

In this section, we conduct a series of experiments to evaluate the effectiveness of our algorithm, and investigate the parameters that affect its performance.

Fig. 10 shows the changing of average service response time during the execution of our algorithm ($|S| = 10$, $|Population| = 50$, $p_c = 0.9$, $p_m = 0.3$). With the increasing of generation, better cache policy with shorter the average service response time is found.

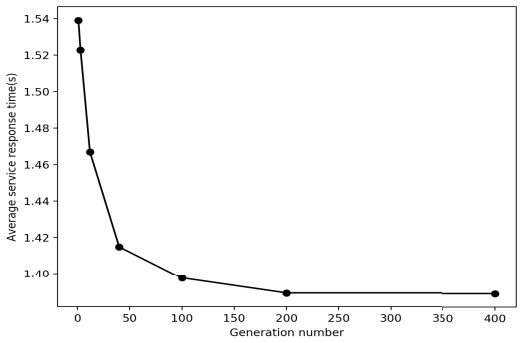

FIGURE 10. Average service response time in different generations.

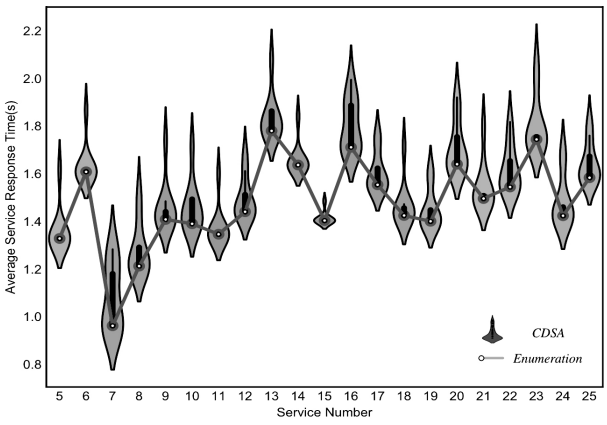

FIGURE 11. Comparison of average service response time.

Fig. 11 shows the ASRT of edge servers when adopting policies generated by the enumeration algorithm and our CDSA model. In this figure, we use a violin plot and a line chart to exhibit the comparison of our algorithm and the ground truth. In the violin plot, the width of each violin shows the diversity of distribution, the thick vertical line in violins shows the scale of data and the hollow circle in violins shows the median of ASRT. In Fig. 11, the CDSA runs 100 times for every given service composition graph with specific service number to make a candidate solution set (as the bodies of violins). We can find that the ASRT varies with the increasing or service number and the CDSA can find the optimal policy that support the request, like using the policy generated by enumeration.

As the CDSA is an optimal evolutionary algorithm based on the constraints of the service provision system, it is necessary to evaluate the results derived from other evolutionary algorithms to check its performance. Fig.12 shows the results of ASRT for the service provision system when given 30 services and their composition relations with different evolutionary optimization algorithms. In our experiment, besides the generic algorithm we also apply other evolutionary algorithms like the particle-swarm optimization (PSO) algorithm and simulated annealing (SA) algorithm to optimize the ASRT of the system. In Table 4 it lists the details about results when running these algorithms 300 times on a provision system with 30 services ($|S| = 14$, $|Population| = 100$,

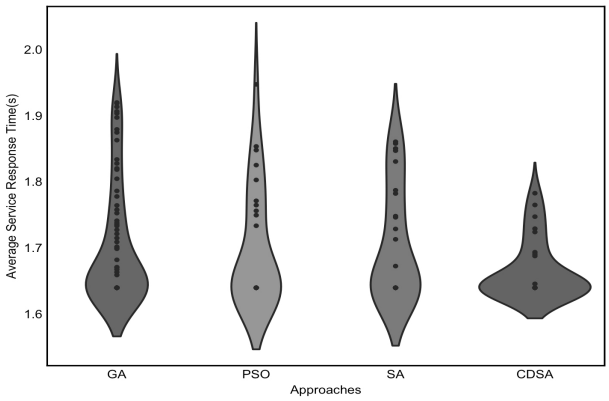

FIGURE 12. Comparison of average service response time.

TABLE 4. Statistics of different approaches.

Approach	min(s)	max(s)	$mean$(s)	$bias(\%)$	opt%
GA	1.639	1.919	1.699	17.08%	53.0%
PSO	1.639	1.946	1.700	18.73%	62.3%
SA	1.639	1.860	1.703	13.48%	55.7%
$CDSA$	1.639	**1.782**	**1.665**	**8.72%**	**73.7%**

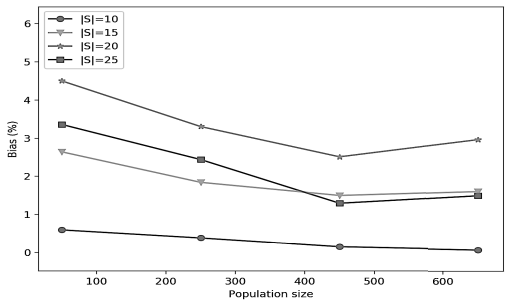

FIGURE 13. Comparison of average service response time.

$p_c = 0.9$, $p_m = 0.2$). From Fig.12 and Table 4, we can find that all these algorithms can find the optimums of the optimization problem. However, the accuracy of them are different. In algorithm GA, PSO and SA, the maximum bias can be 17.08%, 18.73% and 13.48%, while the results of CDSA is 8.72%. This comparison shows that even when algorithms cannot find the optimum, the approximate solution of CDSA will make a cache policy that have 8.72% more ASRT at last. At the same time, the table shows that in the solution set of different algorithms, 73.7% of the solutions in the results of CDSA are equal to the optimum, while those of GA, PSO and SA are 53.0%, 62.3% and 55.7%. This result shows that CDSA has better robustness than others — we have a probability of 73.7% to believe that the result of running CDSA once is the optimal one. It is clear that CDSA performs better than other evolutionary algorithm in this optimization problem. The reason is because that it can always initialized correctly with the resource constraint. With the reduction of search space, it becomes easier for the algorithm to find the optimums.

In Fig. 13, it shows how the parameter $|Population|$ can impact the generated polices of the algorithm. The CDSA is executed 100 times to calculate the bias with ground truth. Although there may be some random factors that impact the result, the trend of the curves can give us some information about the relation. We can find that the bias decreases with the increasing of population size. It means that the algorithm will have higher probability to get the global optimum. This is because it can have more candidate solutions when the population size increases.

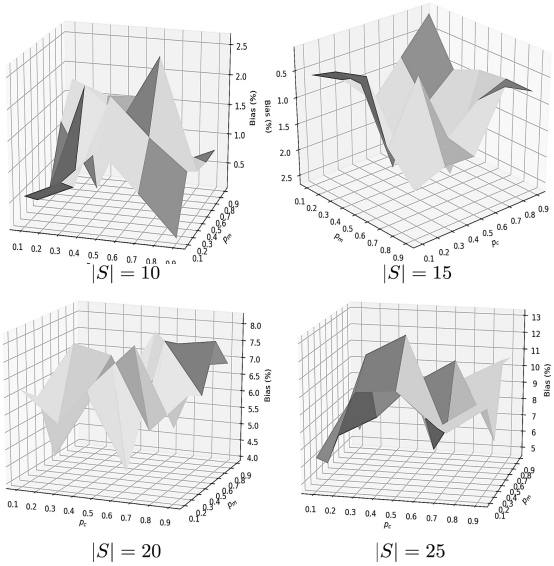

FIGURE 14. Bias for different p_c and p_m.

Fig. 14 shows the impact of parameters p_c and p_m. From this figure we can find that there is no universal p_c and p_m that work well in every scenario. When $|S|$ varies, the best (p_c, p_m) may be different. However, they have something in common to guide us to choose the parameters. The algorithm is suggested to choose a large p_c and a medium p_m. In this way, the algorithm will keep the capability to find global optimums.

B. EFFICIENCY

Since the deployment of services on the cache of edge servers will bring system suspending, it is of vital importance that the efficiency of the deployment algorithm must be high. Thus, in this part, we evaluate the time cost of different algorithms. As the CDSA is made up of 3 parts — encoding, selection and crossover/mutation, the time complexity of it is $O(|S|) + O((n_{population} \times |S|^2 + |S|) \times n_{epoch}) = O(n_{population} \times |S|^2 \times n_{epoch})$. In Fig.15, the curves show the affect of these factors. Comparing the running time of the algorithms and the ground truth in Fig.15(b)(d), we can find that the time cost of enumeration algorithm can be as much as 600s when the service number is only 25 as the points of searching space increase exponentially. On the other hand, the time cost of evolutionary algorithms stays less than 1.6s even when the service number is 50. According to the analysis of the time complexity, the time cost increases linearly as the $n_{population}$ and n_{epoch} increases in Fig.15(a)(c). The result shows that the CDSA is practical in implementation of the service provision system.

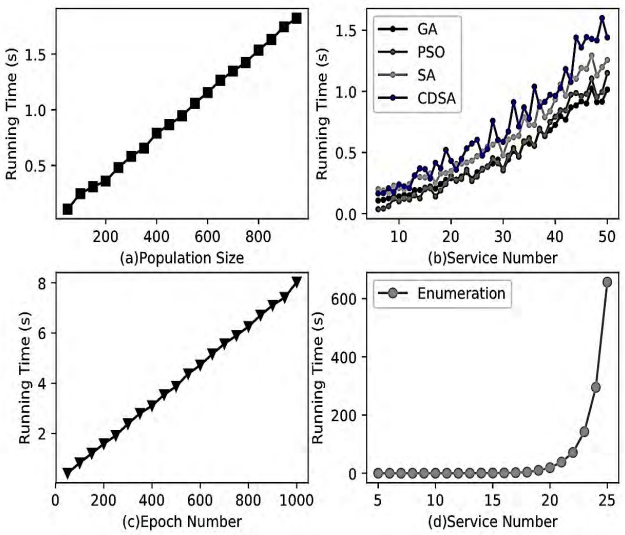

FIGURE 15. Running time of algorithms.

VII. CONCLUSION AND FUTURE WORK

This paper introduces the mobile edge computing model and highlights the cache mechanism with composite services in MEC models. Based on them, we propose a consumption-driven searching algorithm to determine the cache policy. With the help of the cache policy, appropriate services are stored in the edge servers in proximity to support users more better. However, in our model we assume that the frequency or the popularity of services are investigated, which means the policy of deployment shows high dependency on history records, this assumption will result in cold start problem. Besides this, we assume that the execution times on edge server and cloud server are the same, though the assumption will not affect the algorithm but more detailed parameters can help to optimize our model. In future work, we will turn to analyze the latent characteristics of services to help solve this problem.

REFERENCES

[1] A. Giri, S. Dutta, S. Neogy, K. P. Dahal, and Z. Pervez, "Internet of Things (IoT): a survey on architecture, enabling technologies, applications and challenges," in *Proc. 1st Int. Conf. Internet Things Mach. Learn. (IML)*, 2017, Liverpool, U.K., Oct. 2017, pp. 7:1–7:12. [Online]. Available: http://doi.acm.org/10.1145/3109761.3109768

[2] C. Yang, W. Shen, and X. Wang, "Applications of Internet of Things in manufacturing," in *Proc. 20th IEEE Int. Conf. Comput. Supported Cooper. Work Design (CSCWD)*, Nanchang, China, May 2016, pp. 670–675, doi: 10.1109/CSCWD.2016.7566069.

[3] A. Paya and D. C. Marinescu, "Energy-aware load balancing and application scaling for the cloud ecosystem," *IEEE Trans. Cloud Comput.*, vol. 5, no. 1, pp. 15–27, Jan./Mar. 2017.

[4] K. Li, "Improving multicore server performance and reducing energy consumption by workload dependent dynamic power management," *IEEE Trans. Cloud Comput.*, vol. 4, no. 2, pp. 122–137, Apr./Jun. 2016.

[5] S. Deng, H. Wu, W. Tan, Z. Xiang, and Z. Wu, "Mobile service selection for composition: An energy consumption perspective," *IEEE Trans. Autom. Sci. Eng.*, vol. 14, no. 3, pp. 1478–1490, Jul. 2015.

[6] M. Patel *et al.*, "Mobile-edge computing introductory technical white paper," Eur. Telecommun. Standards Inst., Sophia Antipolis, France, White Paper 11, 2014.

[7] X. Larrucea, I. Santamaria, R. Colomo-Palacios, and C. Ebert, "Microservices," *IEEE Softw.*, vol. 35, no. 3, pp. 96–100, May 2018, doi: 10.1109/MS.2018.2141030.

[8] L. Tianze, W. Muqing, Z. Min, and L. Wenxing, "An overhead-optimizing task scheduling strategy for ad-hoc based mobile edge computing," *IEEE Access*, vol. 5, pp. 5609–5622, 2017.

[9] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Trans. Signal Inf. Process. Over Netw.*, vol. 1, no. 2, pp. 89–103, Jun. 2015.

[10] C. You, K. Huang, H. Chae, and B.-H. Kim, "Energy-efficient resource allocation for mobile-edge computation offloading," *IEEE Trans. Wireless Commun.*, vol. 16, no. 3, pp. 1397–1411, Mar. 2017.

[11] S. Wang, Y. Zhao, L. Huang, J. Xu, and C.-H. Hsu, "QoS prediction for service recommendations in mobile edge computing," *J. Parallel Distrib. Comput.*, to be published.

[12] S. Yi, Z. Hao, Q. Zhang, Q. Zhang, W. Shi, and Q. Li, "LAVEA: Latency-aware video analytics on edge computing platform," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Jun. 2017, pp. 2573–2574.

[13] T. Bahreini and D. Grosu, "Efficient placement of multi-component applications in edge computing systems," in *Proc. 2nd ACM/IEEE Symp. Edge Comput.*, 2017, p. 5.

[14] J. Gu, W. Wang, A. Huang, and H. Shan, "Proactive storage at caching-enable base stations in cellular networks," in *Proc. IEEE 24th Int. Symp. Pers. Indoor Mobile Radio Commun. (PIMRC)*, Sep. 2013, pp. 1543–1547.

[15] B. Bai, L. Wang, Z. Han, W. Chen, and T. Svensson, "Caching based socially-aware D2D communications in wireless content delivery networks: A hypergraph framework," *IEEE Wireless Commun.*, vol. 23, no. 4, pp. 74–81, Aug. 2016.

[16] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: Evidence and implications," in *Proc. 18th Annu. Joint Conf. IEEE Comput. Commun. Soc. (INFOCOM)*, vol. 1, Mar. 1999, pp. 126–134.

[17] H. Ahlehagh and S. Dey, "Video-aware scheduling and caching in the radio access network," *IEEE/ACM Trans. Netw.*, vol. 22, no. 5, pp. 1444–1462, Oct. 2014.

[18] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, and D. F. Ferguson, *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More*. Englewood Cliffs, NJ, USA: Prentice-Hall, 2005.

[19] Y. Xu, J. Yin, S. Deng, N. N. Xiong, and J. Huang, "Context-aware QoS prediction for web service recommendation and selection," *Expert Syst. Appl.*, vol. 53, pp. 75–86, Jul. 2016, doi: 10.1016/j.eswa.2016.01.010.

[20] R. Ghosh, A. Ghose, A. Hegde, T. Mukherjee, and A. Mos, "QoS-driven management of business process variants in cloud based execution environments," in *Proc. Int. Conf. Service-Oriented Comput. (ICSOC)* Banff, AB, Canada: Springer, 2016, pp. 55–69.

[21] H. Zhu and I. Bayley, "On the composability of design patterns," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1138–1152, Nov. 2015.

[22] S. Deng, L. Huang, J. Taheri, J. Yin, M. Zhou, and A. Y. Zomaya, "Mobility-aware service composition in mobile communities," *IEEE Trans. Syst., Man, Cybern., Syst.*, vol. 47, no. 3, pp. 555–568, Mar. 2017.

[23] S. Deng, H. Wu, D. Hu, and J. L. Zhao, "Service selection for composition with QoS correlations," *IEEE Trans. Services Comput.*, vol. 9, no. 2, pp. 291–303, Mar. 2016, doi: 10.1109/TSC.2014.2361138.

[24] S. Sahni, "Approximate algorithms for the 0/1 knapsack problem," *J. ACM*, vol. 22, no. 1, pp. 115–124, Jan. 1975.

SHUIGUANG DENG received the B.S. and Ph.D. degrees in computer science from the College of Computer Science and Technology, Zhejiang University, China, in 2002 and 2007, respectively. He was a Visiting Scholar with the Massachusetts Institute of Technology in 2014 and Stanford University in 2015. He is currently a Full Professor with the College of Computer Science and Technology, Zhejiang University. During the past 10 years, he has published more than 100 papers in journals and refereed conferences. His research interests include edge computing, service computing, mobile computing, and business process management. In 2018, he received the Rising Star Award from the IEEE TCSVC. He serves as an Associate Editor for the journal IEEE Access and *IET Cyber-Physical Systems: Theory & Applications*.

ZHENGZHE XIANG received the B.S. degree from Zhejiang University, China, in 2013, where he is currently pursuing the Ph.D. degree with the College of Computer Science. His research interests include service computing, cloud computing, and edge computing.

JIANWEI YIN received the Ph.D. degree in computer science from Zhejiang University (ZJU) in 2001. He was a Visiting Scholar with the Georgia Institute of Technology. He is currently a Full Professor with the College of Computer Science, ZJU. Up to now, he has published more than 100 papers in top international journals and conferences. His current research interests include service computing and business process management. He is an Associate Editor of the IEEE TRANSACTIONS ON SERVICES COMPUTING.

JAVID TAHERI received the Ph.D. degree in mobile computing from the School of Information Technologies, The University of Sydney, Australia. He is currently an Associate Professor with the Department of Computer Science, Karlstad University, Sweden. His major areas of interest are profiling, modeling, and optimization techniques for private and public cloud infrastructures, profiling, modeling, and optimization techniques for software-defined networks, and network-aware scheduling algorithms for cloud and green computing.

ALBERT Y. ZOMAYA (F'04) is currently the Chair Professor of high-performance computing and networking with the School of Information Technologies, The University of Sydney. He published more than 500 scientific papers and articles. His research interests are in the areas of parallel and distributed computing and complex systems. He served as an Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTERS from 2011 to 2014. He serves as an Associate Editor for 22 leading journals, such as the *ACM Computing Surveys*, the IEEE TRANSACTIONS ON COMPUTATIONAL SOCIAL SYSTEMS, the IEEE TRANSACTIONS ON CLOUD COMPUTING, and the *Journal of Parallel and Distributed Computing*.