Offloading Tasks with Dependency and Service Caching in Mobile Edge Computing

Gongming Zhao, *Hongli Xu, *Member, IEEE,* Yangming Zhao, Chunming Qiao, *Fellow, IEEE,* Liusheng Huang, *Member, IEEE,*

Abstract—In Mobile Edge Computing (MEC), many tasks require specific service support for execution and in addition, have a dependent order of execution among the tasks. However, previous works often ignore the impact of having limited services cached at the edge nodes on (dependent) task offloading, thus may lead to an infeasible offloading decision or a longer completion time. To bridge the gap, this paper studies how to efficiently offload dependent tasks to edge nodes with limited (and predetermined) service caching. We formally define the problem of offloading dependent tasks with service caching (ODT-SC), and prove that there exists no algorithm with constant approximation for this hard problem. Then, we design an efficient convex programming based algorithm (CP) to solve this problem. Moreover, we study a special case with a homogeneous MEC and propose a favorite successor based algorithm (FS) to solve this special case with a competitive ratio of $O(1)$. Extensive simulation results using Google data traces show that our proposed algorithms can significantly reduce applications' completion time by about 21-47% compared with other alternatives.

Index Terms—*Mobile Edge Computing, Task Offloading, Service Caching, Dependency, Approximation.*

1 Introduction

The Internet of Things and widespread use of mobile devices are driving the development of many delay-sensitive and resource-intensive applications, such as virtual/augmented reality, face recognition and data stream processing [2] [3] [4]. Currently, these applications are processed or performed on either mobile devices or on a cloud platform. On one hand, mobile devices have too little computational resource for many applications [5]. On the other hand, running resource-intensive applications on a cloud platform often requires massive data be transferred between mobile devices and remote servers in the cloud, leading to unpredictable communication delay [6] [7]. As a result, Mobile Edge Computing (MEC) has emerged as a promising solution to overcome the above disadvantages [8] [9] [10] [11] [12].

However, there still exist many challenges in MEC. We take the face recognition application as an example. Basically, a face recognition application can be divided into five dependent tasks: object acquisition, face detection, preprocessing, feature extraction and classification [13]. When these tasks are offloaded onto edge nodes, we need to take the following factors into considerations.

- *Service caching*. Task execution may require the support of specific services. That means tasks can only be offloaded onto edge nodes configured with corresponding services. For example, tasks "feature extraction" can only be offloaded onto the edge nodes configured with trained machine learning model.
- *Dependency*. There may be dependencies between tasks. For example, the output of task "feature extraction" is the input of task "classification". Thus, task "classification" can start only if task "feature extraction" has completed.

Actually, both *service caching* and *dependency* will impact the feasibility and performance of task offloading [12]. If we do not consider *service caching* or *dependency* when offloading these tasks, the applications may not be performed successfully [3] [14]. Existing works on service caching often focus on the problem of joint optimization of service placement and task offloading in MEC [3] [5] [15] [16]. In fact, service placement/update may incur higher operation cost than task execution, and hurt the system stability [16]. For example, object database and trained machine learning models require a non-trivial amount of data and are time-consuming if we migrate these services [16]. Thus, service placement often occurs at long-term time scale. If we jointly update the service placement and task offloading at long-term time scale (*e.g.*, [5] [15]), due to task dynamics and uncertainty [17] [18], the offloading solutions may lead to computation congestion on some edge nodes. Different from the previous works [3] [5] [15] [16], we assume that services have been placed/cached on edge nodes according to the existing methods such as [5] [15]. We will consider *the impact of service caching on the applications' performance (e.g., the completion time) of dynamic task offloading*.

Due to the limited memory resource, only a subset of services can be cached on an edge node [2]. The status of service caching

- *Some preliminary results of this paper were published in the Proceedings of IEEE INFOCOM 2020 [1].*
- *G. Zhao, *H. Xu (corresponding author) and L. Huang are with the School of Computer Science and Technology, University of Science and Technology of China, Hefei, Anhui, China, 230027, and also with Suzhou Institute for Advanced Study, University of Science and Technology of China, Suzhou, Jiangsu, China, 215123. E-mail: zgm1993@mail.ustc.edu.cn, xuhongli@ustc.edu.cn, lshuang@ustc.edu.cn.*
- *Y. Zhao and C. Qiao are with the Department of Computer Science & Engineering, University at Buffalo, Buffalo, NY, USA, 16260. E-mail:yangming@buffalo.edu, qiao@computer.org.*

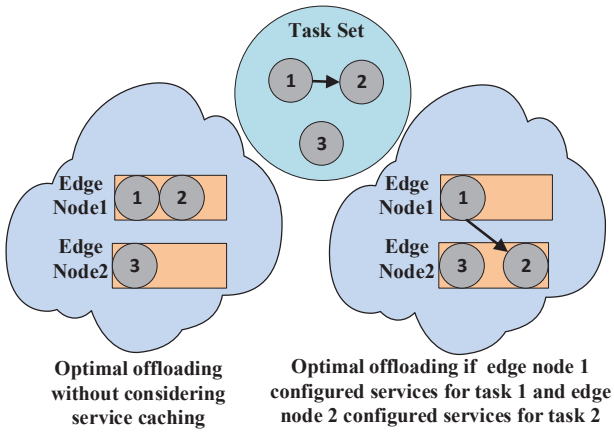


Fig. 1: A Motivation Example. We assume that three tasks need to be offloaded as illustrated in the top plot. The optimal offloading solution is shown in the left plot without considering service caching. The right plot is the offloading solution if only edge node 1 caches services for task 1 and only edge node 2 caches services for task 2.

(*i.e.*, where the services are hosted) will influence the decisions of task offloading. We give an example as shown in Fig. 1. Three tasks need to be offloaded onto two edge nodes. Task 1 must finish and send corresponding data to task 2 before task 2 can start. Task 3 is independent of tasks 1 and 2. For simplicity, the processing delay for any task on any node edge is set as 1 and the communication delay is set as 0.5 for data transmission from task 1 to task 2 if these two tasks are offloaded onto different edge nodes. If we do not consider the constraint of service caching, the offloading solution with the shortest completion time is shown in the left plot of Fig. 1 and the total task completion time (referred to as *makespan* hereafter) is 2. However, if only edge node 1 caches required services for task 1 and only edge node 2 caches required services for task 2, the offloading strategy of the left plot is infeasible/inefficient due to edge node 1 does not cache the services required by task 2. In this case, the feasible solution is shown in the right plot of Fig. 1 and the makespan is 2.5. Thus, this paper focuses on *offloading dependent tasks with service caching*.

We should note that most of the previous work that considered task dependency did not consider the impact of service caching on the task offloading [14] [19] [20], and thus can not be directly applied to the case studied in this work. Work [21] only considered a single edge node that assists a user in executing a sequence of dependent tasks with service caching, so it is difficult to apply to the scenarios with multiple edge nodes. The most related work is GenDoc [22], which jointly considered the problem of dependent task offloading and service caching placement with the objective of application completion time minimization. However, GenDoc does not consider the computing resource constraints on edge nodes when offloading tasks to edge nodes. In fact, mobile edge nodes are resource-sensitive and GenDoc may cause irrational use of limited computing resources.

The main contributions of this paper are as follows:

1) We formally define the problem of offloading dependent tasks in MEC while considering service caching (ODT-SC), and prove its NP-hardness. We also analyze that the ODT-SC problem cannot be solved using a constant approximation algorithm in polynomial time.
2) We present a convex programming based algorithm, called CP, for the ODT-SC problem (*e.g.*, for heterogeneous scenarios). CP transforms this problem into a convex optimization problem and offloads tasks according to the solution of this convex optimization.
3) We design a favorite successor based algorithm, called FS, for the special case (*i.e.*, homogeneous edge nodes), and prove that FS can achieve an approximation ratio of $O(1)$.
4) We conduct extensive simulations using real-world applications (from [23]) and data traces (from [24]) to show that CP and FS help reduce applications' completion time by about 21-47% compared with other alternatives.

The rest of this paper is organized as follows. Section 2 discusses the related works. Section 3 defines the problem of offloading dependent tasks in MEC while considering service caching and proves that there exists no approximation algorithm with a constant factor for this hard problem. In Section 4, we propose an efficient convex programming based algorithm to solve this problem, called CP. Section 5 focuses on the special case of homogeneous edge nodes and proposes an approximation algorithm with bounded approximation ratio. The simulation results are presented in Section 6. We conclude the paper in Section 7.

2 Related Works

The emergence of resource-consuming and delay-sensitive mobile applications, such as 3-D games, augmented reality, and autonomous driving, has spurred a growing need for low-delay access to computing resources [8]. To address these challenges, mobile edge computing (MEC), envisioned as a new computing paradigm, has received an increasing amount of attentions in recent years [2] [8] [9] [10] [11].

In MEC, task offloading is the main research issue in recent years due to its necessarity and importance [8]. Mao *et al.* [10] proposed a low-complexity algorithm to minimize the weighted makespan of multiple independent tasks through jointly optimization of tasks offloading and resources allocation. Tong *et al.* [11] proposed an edge computing architecture according to the distance between the edge nodes and users, and designed an optimal offloading scheme for minimizing the makespan by using a workload placement algorithm. Many works also devoted to minimizing the overall cost of task offloading [25] [26]. Neto *et al.* [25] proposed a lightweight mobile computation offloading framework to minimize overall execution overhead. Huang *et al.* [26] designed a Deep-Q Network based task offloading and resource allocation algorithm to minimize the overall offloading cost in terms of computation cost, energy cost, and delay cost.

The above works assumed that the edge nodes could process any tasks in the system. However, as the mobile applications become increasingly complicated and require the support of various services, the tasks can be processed by these edge nodes with the required services. As a result, since edge nodes cannot be equipped with all services due to limited memory and computing resource

constraints [3] [16], these works can not be applied directly to scenarios with service-aware tasks. This challenge can be solved by considering service caching conditions. Existing works on service caching often focus on the problem of service placement or joint optimization of service placement and task offloading [3] [5] [15] [16]. In fact, service placement/update may incur higher operation cost than task execution, and hurt the system stability [16]. Thus, service placement often occurs at long-term time scale. On the contrary, tasks offloading often occurs at short-term time scale due to task dynamics and uncertainties [17] [18]. That means, making placement/offloading decisions simultaneously may decrease system stability and increase operational cost. Based on this consideration, Farhadi *et al.* [16] separated the time scales of service placement and request scheduling, and proposed a two-time-scale solution for joint optimization of service placement and request scheduling under storage, communication, computation, and budget constraints.

All aforementioned works focus on offloading independent tasks. As modern applications in MEC become increasingly complex, a mobile application may consist of a number of dependent tasks. Thus, offloading dependent tasks is necessary for many practical applications in MEC. Since it is complex by considering precedence constraints and data transfer requirement in MEC, only some works focus on the dependent task offloading problem in MEC such as [14] [19] [20] [21]. Sundar *et al.* [14] proposed a heuristic algorithm to schedule dependent tasks with the objective of overall application execution cost minimization while considering application completion deadline constraints. Hermes *et al.* [19] designed a polynomial time approximation algorithm for offloading dependent tasks to minimize the makespan under resource constraints. Fan *et al.* [20] studied the dependent task offloading problem to minimize the overall cost of all applications with each application's completion time constraints. However, these works do not consider the service caching constraints at the same time.

In fact, many tasks may pose a dependent execution order and in addition, require service support for execution. Thus, considering both service caching and tasks dependency is significant for task offloading. To the best of our knowledge, only a few works considered both two constraints. GenDoc [22] jointly considered the problem of dependent task offloading and service caching placement with the objective of application completion time minimization. However, GenDoc does not consider the processing resource constraints, which may cause irrational use of limited processing resources.

3 Preliminaries and Problem Definition

In this section, we first introduce the system model, including task dependency model and network model. We then formally define the dependent tasks offloading with service caching (ODT-SC) problem, and prove there exists no constant approximation algorithm for this problem.

3.1 System Model

Task Dependency Model: We assume that one or several application(s) (*e.g.*, face recognition, virtual reality) need to be executed at some point in time. These application(s) can be divided into many tasks, each of which can only be executed by one edge node. For each local device that contains task(s), we insert a dummy task as the precedent task of all tasks on this local device. The dummy task must be executed on this local device and the execution time is zero. Note that, task execution may require the support of various resources (*e.g.*, storage, CPU, network I/O) and corresponding services (*e.g.*, machine learning model) [27]. We can restrict the required services so that the dummy task can only be executed on the local device where it is located.

We use $V = \{v_1, v_2, ..., v_n\}$ to denote the set of tasks (including dummy tasks), where $n = |V|$ is the number of tasks. According to Alibaba's data of 4 million applications [22] [28], more than 75% of the applications consist of dependent tasks. That is, modern mobile applications usually contain multiple dependent tasks [14]. For example, a face recognition application can be divided into five dependent tasks: object acquisition, face detection, preprocessing, feature extraction and classification [13]. Given the precedence constraints among these tasks, we use a directed acyclic graph (DAG) $G = (V, E)$ to denote the dependency among tasks, where V denotes the task set and E is the set of edges representing the precedence constraints. More specifically, there is an edge from task v to task v' if and only if there exists data transmission from task v to task v' (*i.e.*, task v' can start only if task v is completed and the corresponding data is transmitted to task v'). We use the parameter $a_{vv'}$ to denote the amount of data that are required to be transferred from task v to task v'. Besides, a sink node of the DAG is a node such that no edge emerges out of it.

Network Model: A typical MEC network contains a set of edge nodes, a remote cloud node and a set of local devices. The cloud node has powerful processing capacity and can cache all services, but it is far away from edge users (local devices), which means that the communication delay of transferring tasks from local devices to the cloud is large. On the contrary, the processing capacity of local devices is weak and only a few services can be cached due to the memory size constraint, but tasks can be executed directly on local devices, i.e., the communication delay can be ignored. For ease of description, we can regard local devices as special edge nodes with low processing capacity, and the cloud as a special edge node with a long transmission distance. We use set $M = \{m_1, m_2, ..., m_l\}$ to represent these execution nodes (including the cloud node and local devices), where $l = |M|$ denotes the number of nodes. These nodes interconnect with each other through various network connections (*e.g.*, local-area network [29]). The communication delay per unit data from nodes m to m' is denoted by $c_{mm'}$ ($c_{mm'} = 0$ if $m = m'$). In this way, if a task is offloaded to an edge node or the remote cloud, we can use the communication delay between this task and the corresponding dummy task to represent the offloading delay of this task.

In MEC, on the one hand, service caching on edge nodes will consume various resources of edge nodes, such as storage and computing resources. On the other hand, compared with the remote cloud, the storage and computing resources of edge nodes are relatively small. For example, the storage space of a small

Symbol	Semantics
V	a set of tasks
E	a set of edges between dependent tasks
M	a set of edge nodes
M_v	a set of edge nodes that meet the services constraints for task $v \in V$
$M_v(R)$	a set of nodes that satisfy both processing resources and services constraints for task $v \in V$
$C(m)$	the processing resource (*e.g.*, CPU cycles) constraints on node $m \in M$
$a_{vv'}$	the amount of data that required to be transferred from task v to task v'
$c_{mm'}$	the communication delay per unit data from node m to node m'
t_{vm}	the execution time if task $v \in V$ is offloaded onto node $m \in M$
r_{vm}	the allocated resources (*e.g.*, storage space) if task v is offloaded onto node m
z_v^m	whether task v is offloaded onto node m
t_v	the start time for executing task $v \in V$
m_v	the offloaded node for executing task $v \in V$

TABLE 1: Key Notations.

mobile base station is about 200GB, and the storage space required for a service is about 20-100GB [3]. As a result, we cannot load all services on each edge node, but only a subset of services. Let M_v represent the set of edge nodes that meet the service constraints for task $v \in V$. That means task $v \in V$ can only be executed by an edge node in M_v. Moreover, each edge node has limited resources (*e.g.*, CPU cycles, storage, computing, I/O [19] [30] [31]). We assume that node $m \in M$ has $C(m)$ resources. According to the attributes of edge nodes and tasks, similar to works [14] [19] [22] [32], through long-term statistics and analysis, if task $v \in V$ is executed on node m, then r_{vm} resources need to be allocated to the task v and the execution time is t_{vm}. For ease of description, we take the resource constraint of the storage space as an example in this paper. Note that, it is easy to extend to multiple resource constraints (*e.g.*, consider both storage, CPU and I/O resource constraints) [19] [31]. Table 1 summarizes some key notations.

3.2 Problem Definition

In MEC, there may contain many applications that need to be processed in time. Given a set of available nodes and a set of applications (each application consists of multiple dependent tasks), we define the problem of offloading these dependent tasks with service caching (ODT-SC). We first construct a DAG according to the dependencies among tasks, as described in Section 3.1. We then use a binary variable z_v^m to denote whether task $v \in V$ is offloaded onto edge node $m \in M$ or not. Let variable t_v denote the start time for task $v \in V$. A feasible offloading solution should satisfy the following conditions:

1) *All Tasks should be Offloaded*: Each task should be offloaded onto exactly one edge node. That means, for each task $v \in V$, $\sum_{m \in M} z_v^m = 1$.
2) *Service Constraint*: Task $v \in V$ can only be offloaded onto the edge node configured with corresponding required services, *i.e.*, the edge node in M_v. That means, $\sum_{m \in M_v} z_v^m = 1, \forall v \in V$.
3) *Dependency Constraint*: For any task pair $\langle v, v' \rangle \in E$, task v' can start iff all precedent tasks are completed and the required data is transferred to the edge node $m_{v'}$. That is, for each task pair $\langle v, v' \rangle \in E$, $t_v + \sum_{m \in M} z_v^m t_{vm} + \sum_{m \in M} \sum_{m' \in M} c_{mm'} a_{vv'} z_v^m z_{v'}^{m'} \leq t_{v'}$.
4) *Execute Tasks in Sequence*: For any pair of tasks $v, v' \in V$, if both tasks are offloaded onto the same edge node $m \in M$ (*i.e.*, $m_v = m_{v'} = m$), then $t_v + t_{vm} \leq t_{v'}$ or $t_{v'} + t_{v'm} \leq t_v$. That means, each edge node can only perform one task at a time instance and tasks cannot be interrupted during the execution [14].
5) *Processing Resource Constraints*: The processing resource constraint should be satisfied for every edge node, which can be formulated as $\sum_{v \in V} z_v^m r_{vm} \leq C(m), \forall m \in M$.

The makespan of these tasks is denoted by $T = \max\{t_v + \sum_{m \in M} z_v^m t_{vm}, v \in V\}$. We aim to find a feasible offloading solution with a minimum makespan. Thus, the ODT-SC problem can be formulated as follows:

$$\min \quad T$$
$$s.t. \begin{cases} \sum_{m \in M} z_v^m = 1, & \forall v \in V \\ \sum_{m \in M_v} z_v^m = 1, & \forall v \in V \\ t_v + \sum_{m \in M} z_v^m t_{vm} + \sum_{m \in M} & \\ \sum_{m' \in M} c_{mm'} a_{vv'} z_v^m z_{v'}^{m'} \leq t_{v'}, & \forall \langle v, v' \rangle \in E \\ \text{If } t_v \geq t_{v'} \text{ and } z_v^m = z_{v'}^m = 1 & \\ \quad \text{then: } t_v - t_{v'} \geq t_{v'm}, & \forall v, v' \in V, m \in M \\ \sum_{v \in V} z_v^m r_{vm} \leq C(m), & \forall m \in M \\ t_v + \sum_{m \in M} z_v^m t_{vm} \leq T, & \forall v \in V \\ t_v \geq 0, & \forall v \in V \\ z_v^m \in \{0, 1\}, & \forall m \in M, v \in V \end{cases} \tag{1}$$

The first set of equations represents that each task should be offloaded onto exactly one edge node. The second set of equations denotes the services constraint, *i.e.*, task $v \in V$ can only be offloaded onto the edge node in set M_v. The third set of inequalities represents the dependency constraint, *i.e.*, task v' can start iff all precedent tasks are completed and transferred corresponding data to the task v'. The fourth set of constraints guarantees that all tasks on the same edge node will be executed in sequence. More specifically, if tasks $v \in V$ and $v' \in V$ are offloaded onto the same edge node $m \in M$ and without loss of generality we assume task v starts later than task v', *i.e.*, $t_v \geq t_{v'}$. In this case, we need to ensure that task v can start only if task v' is completed, *i.e.*, $t_v - t_{v'} \geq t_{v'm}$. The fifth set of inequalities represents the processing resource constraints. Our objective is to minimize the maekspan, *i.e.*, $\min T$.

3.3 Complexity Analysis

In this section, we prove that ODT-SC is one of the most difficult NP-hard problems for which there exists no approximation

algorithm with a constant factor.

Theorem 1. ODT-SC is one of the most difficult problems in NP-hard class: even finding a k-approximation algorithm (k is a constant) to solve ODT-SC is NP-hard.

To prove this theorem, we first give the following definition.

Definition 1 (Travelling Salesman Problem (TSP) [33]). Given a set of cities and the distance between every pair of cities, the problem is to find the shortest route on which each city is visited exactly once and return to the starting point. In other words, given an undirected complete graph $G(B, A)$ and the weight of each edge in A, the objective is to find an optimal Hamiltonian cycle.

Proof: In the following, we first prove that TSP is a special case of the ODT-SC problem and then show that finding a k-approximation algorithm for TSP is NP-Hard.

We consider an arbitrary TSP instance Λ. There are a set of cities $C = \{1, 2, ..., h\}$, with $h = |C|$, and the distance is denoted by d_{ij} between each pair of cities $i, j \in C$. Now, we construct a special case of the ODT-SC problem. Assume there are h identical edge nodes equipped with all required services, denoted by $M = \{1, 2, ..., h\}$, which are one-to-one correspondence with the cities in set C. The available processing resources of edge node $m \in M$ are the same, denoted by α. $h+1$ tasks are required to be offloaded onto these h edge nodes, denoted by $V = \{v_1, v_2, ..., v_{h+1}\}$, and the DAG for the tasks is : $v_1 \rightarrow v_2 \rightarrow ... \rightarrow v_{h+1}$. The execution time $t_{vm} = t$ for any task $v \in V$ on any edge node $m \in M$. Task v_1 and task v_{h+1} require $\alpha/2$ processing resources, respectively, while the other tasks require α processing resources each. The data volume that required to be transmitted $a_{vv'} = 1$ for each edge $\langle v, v' \rangle$ on the DAG. The communication delay per unit data between edge nodes $i, j \in M$ is equal to the distance between cities i and j (*i.e.*, c_{ij} = d_{ij}). The objective is to find a feasible offloading with a minimum makespan. For this special case, obviously, tasks v_1 and v_{h+1} will share one edge node while each of the other edge nodes will execute only one task. The start time $t_{v_i} = t_{v_{i-1}} + t + c_{m_{v_{i-1}} m_{v_i}}$ for each task $v_i \in V - \{v_1\}$ and $t_{v_1} = 0$. Thus, we can obtain the maskspan:

$$\begin{aligned} T &= t_{v_{h+1}} + t = t_{v_h} + t + d_{m_{v_h} m_{v_{h+1}}} + t \\ &= (h+1)t + d_{m_{v_1} m_{v_2}} + d_{m_{v_2} m_{v_3}} + ... + d_{m_{v_h} m_{v_{h+1}}} \end{aligned} \quad (2)$$

That means, for each task $v_i \in V$, the selection of edge node m_{v_i} turns into the selection of ith visited city. This is exactly the TSP instance Λ. Thus, each TSP instance is a special case of the ODT-SC problem.

Previous works have proved that finding a k-approximation (k is a constant) algorithm for TSP is NP-Hard [33] [34]. We give a briefly proof for completeness. Let $G_1(B, A_1)$ be any graph, where B denotes the vertex set and A_1 denotes the edge set. Let $b = |B|$ represent the number of vertices. We construct the complete graph $G(B, A)$ such that $A = \{\langle p, q \rangle \, | p, q \in B\}$ and define the weight/length of each edge $\langle p, q \rangle \in A$ as:

$$d_{pq} = \begin{cases} 1, & \langle p, q \rangle \in A_1 \\ kb, & otherwise \end{cases} \quad (3)$$

The TSP problem is to find a shortest Hamiltonian cycle in graph $G(B, A)$. Assuming there exists a k-approximation algorithm, we denote the optimal solution as OPT_{HC} and the approximation solution as A_{HC}. Obviously, there exists Hamiltonian cycle in $G_1(B, A_1)$ if and only if:

$$A_{HC} \leq k \cdot OPT_{HC} = kb \quad (4)$$

These is no Hamiltonian cycle in $G_1(B, A_1)$ if and only if:

$$A_{HC} \geq OPT_{HC} \geq kb + b - 1 > kb \quad (5)$$

Consequently, if the solution $A_{HC} \leq kb$, then Hamiltonian cycle exists in $G_1(B, A_1)$. If the solution $A_{HC} > kb$, then there is no Hamiltonian cycle in $G_1(B, A_1)$. That means, we can judge whether there is Hamiltonian cycle in any graph $G_1(B, A_1)$ according to the solution of the approximation algorithm (*i.e.*, in polynomial time). However, the Hamiltonian cycle problem is NP-complete, which cannot be solved in polynomial time [35] unless $P = NP$. Therefore, the assumption is false. As a result, finding a k-approximation algorithm for TSP is NP-Hard. Considering that TSP is a special case of ODT-SC, we can conclude that finding a k-approximation algorithm (k is a constant) to solve ODT-SC is NP-hard. □

The above analysis shows the hardness of the ODT-SC problem. Thus, in this paper, we first design algorithms to solve the general ODT-SC problem in Section 4 and then design an approximation algorithm with bounded approximation factor for the homogenous scenario in Section 5.

4 Algorithms Design for ODT-SC

We first propose a rounding based algorithm to solve ODT-SC in Section 4.1. Although this method is non-trivial, it cannot provide satisfactory performance as shown in simulation section. Then, we propose convex programming based algorithm (CP), which will be described in Section 4.2.

4.1 Rounding based Algorithm

In this section, we propose a rounding based algorithm for ODT-SC, called Rounding. Specifically, the Rounding algorithm offloads dependent tasks through three major steps: 1) *relaxing the constraints of ODT-SC* for computing the potential edge node m_v and execution time t_v for each task $v \in V$; 2) *determining the scheduling order of tasks* to meet the dependency constraints; and 3) *edge node selection* for scheduling each task in order with the aim of minimizing the makespan.

Relaxing the Constraints of ODT-SC. By Eq. (1), ODT-SC is very difficult to be solved directly. Specifically, the third set of inequalities in Eq. (1) is quadratic and the fourth set of constraints in Eq. (1) contains conditional statement. To eliminate these difficulties, we first define binary variables $u_{vv'}^{mm'}$ for any two tasks $v, v' \in V$ and any two edge nodes $m, m' \in M$ that satisfy:

$$\frac{z_v^m + z_{v'}^{m'} - 1}{2} \leq u_{vv'}^{mm'} \leq \frac{z_v^m + z_{v'}^{m'}}{2}, \forall v, v' \in V, m, m' \in M \quad (6)$$

Considering that z_v^m and $z_{v'}^{m'}$ are both binary variables, it follows $z_v^m \cdot z_{v'}^{m'} = u_{vv'}^{mm'}$ and we can modify the third set of inequalities in Eq. (1) as follows:

$$\begin{aligned} & t_v + \sum_{m\in M} z_v^m t_{vm} + \sum_{m\in M}\sum_{m'\in M} c_{mm'} a_{vv'} z_v^m z_{v'}^{m'} \\ =& t_v + \sum_{m\in M} z_v^m t_{vm} + \sum_{m\in M}\sum_{m'\in M} c_{mm'} a_{vv'} u_{vv'}^{mm'} \\ \leq& t_{v'}, \forall \langle v, v'\rangle \in E \end{aligned} \tag{7}$$

We then let χ represent a large number and $x_{vv'} \in \{0,1\}, \forall v, v' \in V$. In this way, we can modify the fourth set of constraints in Eq. (1) as the following inequalities:

$$\frac{t_v - t_{v'}}{\chi} < x_{vv'}, \forall v, v' \in V \tag{8}$$

$$\chi(3 - z_v^m - z_{v'}^m - x_{vv'}) + t_v - t_{v'} \geq t_{v'm}, \forall v, v' \in V, m \in M \tag{9}$$

More specifically, for any two tasks v and v', there are two cases: 1) Both tasks v and v' are offloaded onto the same edge node $m \in M$. Without loss of generality, we assume that task v starts later than task v', *i.e.*, $t_v \geq t_{v'}$. In this case, both z_v^m and $z_{v'}^m$ are equal to 1 and Eq. (8) guarantees $x_{vv'} = 1$. Thus, $\chi(3 - z_v^m - z_{v'}^m - x_{vv'}) = 0$ and Eq. (9) can be simplified to $t_v - t_{v'} \geq t_{v'm}$, which guarantees that task v cannot start before task v' is finished. 2) Tasks v and v' are offloaded onto different edge nodes, which means z_v^m and $z_{v'}^m$ cannot be equal to 1 simultaneously. Under this case, $3 - z_v^m - z_{v'}^m - x_{vv'}$ is larger than 0 and Eq. (9) holds regardless of the values of t_v and $t_{v'}$ (*i.e.*, there is no constraint between t_v and $t_{v'}$). Thus, the above two sets of inequalities guarantee that all tasks on the same edge node will be executed in sequence and tasks offloaded onto different edge nodes can be performed simultaneously if necessary. It means that we transform the fourth set of constraints in Eq. (1) into the above two sets of inequalities. In the end, we relax all binary variables. Specifically, ODT-SC assumes that each task can only be performed onto exact one edge node. By relaxing this assumption, each task $i \in V$ is permitted to be splittable and performed onto several edge nodes. To sum up, we formulate the problem as Eq. (10).

Since Eq. (10) is a linear program, we can solve it in polynomial time with a linear program solver such as PuLP [36]. Assume that the optimal solutions for Eq. (10) are denoted by $\check{z}_v^m$, $\check{t}_v$, $\check{x}_{vv'}$ and $\check{u}_{vv'}^{mm'}$, $\forall v, v' \in V, \forall m, m' \in M$, and the optimal result is denoted by $\check{T}$. As Eq. (10) is a relaxation of the ODT-SC problem, $\check{T}$ is a lower-bound result for this problem. According to these optimal solutions obtained by Eq. (10), we can get the potential edge node and execution time for each task $v \in V$, which will be used for the following operations.

$$\min \quad T$$

$$s.t. \begin{cases} \sum_{m\in M} z_v^m = 1, & \forall v \in V \\ \sum_{m\in M_v} z_v^m = 1, & \forall v \in V \\ \frac{z_v^m + z_{v'}^{m'} - 1}{2} \leq u_{vv'}^{mm'} \leq \frac{z_v^m + z_{v'}^{m'}}{2}, & \forall v, v' \in V, m, m' \in M \\ t_v + \sum_{m\in M} z_v^m t_{vm} + \sum_{m\in M} & \\ \sum_{m'\in M} c_{mm'} a_{vv'} u_{vv'}^{mm'} \leq t_{v'}, & \forall \langle v, v'\rangle \in E \\ \frac{t_v - t_{v'}}{\chi} < x_{vv'}, & \forall v, v' \in V \\ \chi(3 - z_v^m - z_{v'}^m - x_{vv'}) + & \\ t_v - t_{v'} \geq t_{v'm}, & \forall v, v' \in V, m \in M \\ \sum_{v\in V} z_v^m r_{vm} \leq C(m), & \forall m \in M \\ t_v + \sum_{m\in M} z_v^m t_{vm} \leq T, & \forall v \in V \\ t_v \geq 0, & \forall v \in V \\ z_v^m \in [0,1], & \forall m \in M, v \in V \\ x_{vv'} \in [0,1], & \forall v, v' \in V \\ u_{vv'}^{mm'} \in [0,1], & \forall v, v' \in V, m, m' \in M \end{cases} \tag{10}$$

Determining the Execution Order of Tasks. We sort all tasks by the increasing order of their starting time in a scheduling list Π. Tie-breaking is done randomly for simplicity. Based on the fourth set of inequalities in Eq. (10), it can be easily shown that the increasing order of $\check{t}_v$ for $v \in V$ preserves the dependency constraints. Thus, we schedule tasks one by one in the order of scheduling list Π in the next step to preserve the dependency constraints.

Edge Nodes Selection. We offload task $v \in V$ to edge node $m \in M$ based on the the optimal solution $\check{z}_v^m$ using the randomized rounding method [37]. More specifically, for each task $v \in V$, we choose one edge node $m \in M$ to set $z_v^m = 1$ and set the other values as 0, which means that we will offload task v to edge node m, with the probability of $\check{z}_v^m$. Thus, for each task v in scheduling list Π, we offload task v to edge node $m \in M$ with the probability of $\check{z}_v^m$. In this way, we can determine the offloading schedule for all tasks. However, due to the randomized rounding processing, the selected edge node $m \in M$ for task $v \in V$ may break processing resource constraints.

4.2 Convex Programming based Algorithm

In this section, we present a convex programming based algorithm for ODT-SC, called CP. The workflow of CP is shown in Fig. 2. Similar to Rounding, CP offloads dependent tasks through four major steps: 1) *Relaxing the ODT-SC problem* to construct a convex optimization program. 2) *Using progressive rounding* method to obtain feasible solutions. 3) *Computing weight* for each task according to the feasible solutions. 4) *Offloading tasks* based on the weight values.

Relaxing the ODT-SC Problem. We find that the Rounding algorithm cannot achieve good results due to introducing too many variables. Thus, in this section, to eliminate the complexity of Eq. (1), we first leverage the definition of binary variable z_v^m, with $\forall v \in V$ and $\forall m \in M$, to give the following modification:

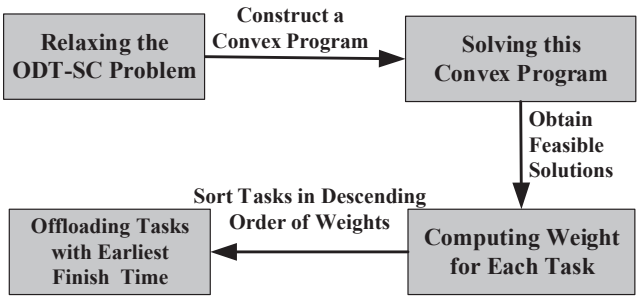


Fig. 2: Workflow of the CP algorithm. CP can be divided into four steps: relaxing the ODT-SC problem to construct convex program, using progressive rounding method to solve this program, computing weight for each task according to the solutions and offloading tasks according to the weights.

$$t_v + \sum_{m\in M} z_v^m t_{vm} + \sum_{m\in M}\sum_{m'\in M} c_{mm'}a_{vv'}z_v^m z_{v'}^{m'} = t_v + \sum_{m\in M} z_v^m t_{vm} + \sum_{m\in M}\sum_{m'\in M} c_{mm'}a_{vv'}\max[z_v^m + z_{v'}^{m'} - 1, 0] \tag{11}$$

Then we assume that edge nodes can perform tasks in parallel and each task $v \in V$ is permitted to be splittable and can be executed on several edge nodes. In this way, we derive the following convex optimization problem:

$$\min \quad T$$

$$s.t. \begin{cases} \sum\limits_{m\in M} z_v^m = 1, & \forall v \in V \\ \sum\limits_{m\in M_v} z_v^m = 1, & \forall v \in V \\ t_v + \sum\limits_{m\in M} z_v^m t_{vm} + \sum\limits_{m\in M}\sum\limits_{m'\in M} c_{mm'} & \\ a_{vv'}\cdot\max[z_v^m + z_{v'}^{m'} - 1, 0] \le t_{v'}, & \forall \langle v, v'\rangle \in E \\ \sum\limits_{v\in V} z_m^v r_{vm} \le C(m), & \forall m \in M \\ t_v + \sum\limits_{m\in M} z_v^m t_{vm} \le T, & \forall v \in V \\ t_v \ge 0, & \forall v \in V \\ z_v^m \in [0,1], & \forall m \in M, v \in V \end{cases} \tag{12}$$

Since Eq. (12) is a convex optimization problem, we can solve it in polynomial time with a convex programming solvers such as CPLEX [38]. Assume that the optimal solutions for Eq. (12) are $\widetilde{z}_v^m$ and $\widetilde{t}_v$, $\forall v \in V, m \in M$, and the optimal objective value is $\widetilde{T}$.

Progressive Rounding. This step obtains integer solutions $\widehat{z}_v^m$ for each $v \in V$ and $m \in M$ using the progressive rounding method [39]. More specifically, in each iteration, we first solve Eq. (12) and obtain fractional solutions $\widetilde{z}_v^m$. Then we choose some tasks $v \in V$ with large value of $\max\{\widetilde{z}_v^m, m \in M\}$ and use randomized rounding method [37] to derive an integer solution $\widehat{z}_v^m$ for these chosen tasks. We fix the integer solution in $\widehat{z}_v^m$ (*i.e.*, fix these rounding solutions as known quantities) and solve Eq. (12) again. In this way, we can obtain a feasible solutions $\widehat{z}_v^m$ after several iterations. That means, we get the offloaded edge node m_v for each task $v \in V$ (*i.e.*, $\widehat{z}_v^{m_v} = 1$) while assuming edge node can simultaneously execute tasks in this step.

Computing Weights. This step computes the weight for each task. More specifically, we first insert an end task at the bottom of the DAG and connect this task with all sink nodes of the DAG. We then denote the weight of each link $\langle v, v'\rangle \in E$ as $w\langle v, v'\rangle = t_{vm_v} + c_{m_v m_{v'}} a_{vv'}$ and the weights of links connected with the end task are set as 0. In this way, we compute the maximum distance from each task $v \in V$ to the end task, denoted by $W(v)$. It can be easily shown that the descending order of their distance to the end task preserves the dependency constraints and a larger value means potential longer execution time. Thus, we sort all tasks in descending order of their distance to the end task and keep them in a list Π.

Offloading Tasks. We offload tasks following the order of list Π in this step. We define some concepts/variables to facilitate the description of this part. We use $Pred(v)$ and $Succ(v)$ to denote the set of immediate predecessor and successor tasks of task $v \in V$, respectively, which can be obtained according to the DAG. Let $R(m)$ denote the rest processing resources for edge node $m \in M$, initialized as $C(m)$. $M_v(R)$ denotes the set of edge nodes that satisfy both processing resource and service constraints for task $v \in V$, formally $M_v(R) = M_v \bigcap \{m \,|\, R(m) \ge r_{vm}, m \in M\}$. Let $T(v, m)$ represent the first date at which there is a larger idle time slot on edge node $m \in M$ than t_{vm}, initialized as 0. Note that, the idle time slot may be between two already-offloaded tasks on edge node m or after the time all tasks offloaded onto edge node m are completed. Moreover, for each task $v \in V$, the actual offloaded edge node, the actual start time and the actual finish time are denoted by m_v, $\bar{t_v}$ and $\bar{f_v}$, respectively, and are initialized to 0. With these definitions, if task v is offloaded onto edge node m, we can calculate the earliest start time $EST(v, m)$ and the earliest finish time $EFT(v, m)$:

$$EST(v,m) = \max(T(v,m), \max_{v'\in Pred(v)}(\bar{f_{v'}} + c_{m_{v'}m}a_{v'v})) \tag{13}$$

$$EFT(v,m) = EST(v,m) + t_{vm} \tag{14}$$

In each iteration, we choose the first task v in list Π for offloading. We first compute the earliest finish time $EFT(v, m)$ for each edge node $m \in M_v(R)$ by Eq. (14) and then offload task v onto edge node m with $\min_{m\in M_v(R)} EFT(v, m)$. After task v is offloaded, we record the actual offloaded edge node m_v. Besides, we update the actual start time $\bar{t_v}$, the actual finish time $\bar{f_v}$ and the rest processing resources $R(m_v)$ by Eqs. (15), (16) and (17), respectively.

$$\bar{t_v} = EST(v, m_v) \tag{15}$$

$$\bar{f_v} = \bar{t_v} + t_{vm_v} \tag{16}$$

$$R(m_v) = R(m_v) - r_{vm} \tag{17}$$

We repeat these operations until all tasks have been offloaded. The CP algorithm is formally described in Algorithm 1.

Algorithm 1 CP: Convex Programming based Algorithm for ODT-SC

Step 1: Relaxing ODT-SC Problem
Construct a convex optimization program in Eq. (12)
Obtain the optimal solution $\widetilde{z}_v^m$, $\widetilde{t}_v$
Step 2: Progressive Rounding
Derive an integer solution $\widehat{z}_v^m$ by progressive rounding for each $v \in V$, $m \in M$
Step 3: Computing Weights
Compute $W(v)$ for each task $v \in V$
Sort all tasks $v \in V$ in descending order of their distance to the end task and saved in list Π
Step 4: Offloading Tasks
for each task $v \in V$ **do**
    Obtain $Pred(v)$ according to DAG
    Initialize variables $\bar{t_v}$ and $\bar{f_v}$ to 0
end for
for each edge node $m \in M$ **do**
    Initialize variables $R(m)$ to $C(m)$
end for
while offloading list $\Pi \neq \emptyset$ **do**
    Select the first task v from the list Π
    Update $M_v(R) = M_v \bigcap \{m \,|\, R(m) \geq r_{vm},\, m \in M\}$
    Compute $EFT(v, m)$ with Eq. (14) for $m \in M_v(R)$
    Offload task v on m with $\min_{m \in M_v(R)} EFT(v, m)$
    Record the offloaded edge node as m_v
    Update $\bar{t_v}$, $\bar{f_v}$ and $R(m_v)$ with Eq. (15), Eq. (16) and Eq. (17), respectively
    Delete task v from offloading list Π
end while

5 Favorite Successor based Algorithm for the Practical Homogeneous Scenario

The above section has proposed the CP algorithm to solve the general ODT-SC problem (*i.e.*, heterogeneous scenario). In many practical scenarios, since most of the edge nodes are placed/purchased at the same time by providers, the hardware specifications of edge nodes are similar [40] [41] [42]. Thus, this section studies the offloading problem in homogeneous scenarios. We assume that all edge nodes have similar processing capacity and all links have similar transmission rate. In other words, we use e_v to denote the execution time t_{vm} if task $v \in V$ is offloaded onto edge node $m \in M$ (*i.e.*, $e_v = t_{vm}$) and use c to denote the communication delay per unit data $c_{mm'}$ for each pair of edge nodes $m, m' \in M$ (*i.e.*, $c = c_{mm'}$).

MEC decreases the task offloading delay by placing computing resources in close proximity to the local devices. On the one hand, with the development of the 5G technology, the data transmission rate has been greatly improved [3]. On the other hand, the processing capacity of edge nodes is limited. Thus, for the typical applications, *e.g.*, virtual/augmented reality (VR/AR), cognitive assistance and mobile gaming, the task execution delay is much greater than the transmission delay [2]. Thus, in the practical scenarios discussed in this section, we assume the minimum processing delay is greater than the maximum communication delay [41]. This section presents an approximate algorithm with bounded approximation factor for offloading tasks in the practical homogeneous scenarios.

5.1 Favorite Successor based Algorithm Description

In a dependent task set V, each task $v \in V$ may have several precedent tasks (*i.e.*, predecessors) and several succedent tasks (*i.e.*, successors). The predecessor/successor of task v that offloaded onto the same edge node as task v does not consume communication delay for data transmission between different edge nodes. Thus, how to select predecessor/successor of task v to offload onto the same edge node as task v is important during scheduling dependent tasks. Based on this consideration, we first give the definitions of favorite successor and predecessor for this problem.

Definition 2 (Favorite Successor [43]). For any task $v \in V$, if task $v' \in Succ(v)$ satisfies $\bar{t_{v'}} < \bar{t_v} + e_v + ca_{vv'}$, then task v' is called the favorite successor for task v.

Definition 3 (Favorite Predecessor [43]). For any task $v \in V$, if task $v' \in Pred(v)$ satisfies $\bar{t_{v'}} + e_{v'} + ca_{v'v} > \bar{t_v}$, then task v' is called the favorite predecessor for task v.

According to the above definitions, we prove that each task $v \in V$ has at most one favorite successor/predecessor and the favorite successor/predecessor v' must be offloaded onto the same edge node as task v. Specifically, if there are two favorite successors v' and v'' of task v, then both tasks v' and v'' should be offloaded to the same edge node as task v. Otherwise $\bar{t_{v'}}(\bar{t_{v''}}) \geq \bar{t_v} + e_v + c \cdot a_{vv'}(c \cdot a_{vv''})$, which contradicts the definition of favorite successor. Without loss of generality, assume that $\bar{t_{v'}} \geq \bar{t_{v''}}$. In this way, $\bar{t_{v'}}$ will be performed at least at time $\bar{t_v} + e_v + e_{v''} \geq \bar{t_v} + e_v + c \cdot a_{vv'}$. That is, $\bar{t_{v'}} \geq \bar{t_v} + e_v + c \cdot a_{vv'}$, which contradicts the definition of favorite successor. Thus, each task $v \in V$ has at most one favorite successor. Similarly, we can prove that each task has at most one favorite predecessor.

Based on the definition of favorite successor, we present a favorite successor based algorithm to solve the special case (FS). Specifically, FS offloads tasks through two major steps: 1) *obtain favorite successor* without considering services and the number of edge nodes constraints. The results can reflect the dependency priority of the DAG. 2) *favorite successor based offloading* while considering the number of edge nodes and service constraints.

Obtaining Favorite Successor. We first attempt to offload tasks to edge nodes without considering services and the number of edge nodes constraints. In this situation, we only need to consider the dependency constraint and we formulate this problem as follows:

$$\min \quad T$$

$$s.t.\begin{cases} t_v + e_v + ca_{vv'}y_{vv'} \le t_{v'}, & \forall \langle v, v' \rangle \in E \\ \sum_{v' \in Succ(v)} y_{vv'} \ge |Succ(v)| - 1, & \forall v : \langle v, v' \rangle \in E \\ \sum_{v' \in Pred(v)} y_{v'v} \ge |Pred(v)| - 1, & \forall v : \langle v', v \rangle \in E \\ t_v + e_v \le T, & \forall v \in V \\ t_v \ge 0, & \forall v \in V \\ y_{vv'} \in \{0, 1\}, & \forall \langle v, v' \rangle \in E \end{cases} \tag{18}$$

where binary variable $y_{vv'}$ denotes whether task v' is the favorite successor of task v. Specifically, $y_{vv'} = 0$ represents that task v' is the favorite successor of task v, otherwise $y_{vv'} = 1$. The first set of inequalities indicates the dependency constraints. Specifically, if $y_{vv'} = 0$, then this set of inequalities turns into $t_v + e_v \le t_{v'}$ (*i.e.*, no communication delay between task v and task v' if v' is the favorite successor of v). Otherwise, it turns into $t_v + e_v + ca_{vv'} \le t_{v'}$ (*i.e.*, we need consider communication delay between task v and task v' if v' is not the favorite successor of v). The second and third sets of inequalities indicate that any task has at most one favorite successor/predecessor. For example, if task $v \in V$ has more than one favorite successor, then $\sum_{v' \in Succ(v)} y_{vv'} < |Succ(v)| - 1$, which contradicts the second inequality. The fourth set of inequalities means the task offloading delay, *i.e.*, task v can be executed only when it has been offloaded from the local device to the edge node. Our objective is to minimize the makespan, *i.e.*, min T.

To solve this program in polynomial time, we relax the sixth set of constraints by setting $y_{vv'} \in [0, 1]$. In this way, we can obtain the optimal solutions $\widetilde{y}_{vv'}$ ($\forall \langle v, v' \rangle \in E$) with a linear programming solver such as PuLP [36]. The second set of inequalities indicates that at most one successor v' of task $v \in V$ can satisfy $\widetilde{y}_{vv'} < 0.5$. Specifically, if there exists two successors v' and v'' of task $v \in V$ such that $\widetilde{y}_{vv'} < 0.5$ and $\widetilde{y}_{vv''} < 0.5$, then we have $\sum_{v' \in Succ(v)} y_{vv'} < |Succ(v)| - 1$, which contradicts with the second set of inequalities. Thus, for each $\langle v, v' \rangle \in E$, let $\widehat{y}_{vv'} = 0$ if $\widetilde{y}_{vv'} < 0.5$, and let $\widehat{y}_{vv'} = 1$ otherwise. In this way, we get the integer solutions, that is, the favorite successor (if exists) for each task.

Favorite Successor based Offloading. In this step, we leverage the results obtained by the first step and the list offloading algorithm [23] to offload tasks. That is, we offload tasks one by one with the earliest start time while trying to assign the favorite successor of task i to the same edge node as task i. During the offloading process, we first introduce some definitions for the sake of convenience. We use V_R to denote the set of tasks whose all predecessors have been offloaded, initialized as the set of tasks without predecessor. In other words, V_R denote the set of tasks that can be offloaded at this time. Moreover, let l_m denote the last offloaded task on edge node $m \in M$ at this time and f_m denote the favorite successor of task l_m, both initialized as none. We use $Avail(v) = \max_{v' \in Pred(v)}(\bar{t_{v'}} + e_{v'} + ca_{v'v})$ to denote the available time that task v can be executed on any edge node. That is, task v can be executed only after all its predecessors have been executed and the corresponding data has been transmitted.

Then we compute the earliest start time $EST(v, m)$ with Eq. (13) for each task $v \in V_R$ and edge node $m \in M_v$. For each edge node $m \in M_{f_m}$, if $EST(f_m, m) < T(f_m, m) + ca_{l_m f_m}$, edge node m can execute the favorite successor f_m earlier than other edge nodes. Thus, we try to reserve edge node m for executing task f_m. For each task $v \in V_R \bigcap Succ(l_m)$ and $v \ne f_m$ and $m \in M_v$ (*i.e.*, v is a potential competing successor), if 1) $Avail(v) \ge EST(f_m, m)$ or 2) there is an edge node $m' \ne m$ such that $EST(v, m') \le Avail(v)$ and if $m' \in M_{f_{m'}}$ and $EST(f_{m'}, m') < T(f_{m'}, m') + ca_{l_{m'} f_{m'}}$, $v \notin Succ(l_{m'})$ or $v = f_{m'}$, we defer the earliest staring time of task v on edge node m, that is, $EST(v, m) = EST(f_m, m) + e_{f_m}$. Then we choose edge node m' with $\min_{v' \in V_R, m' \in M_{v'}} EST(v', m')$ to offload task v'. After task v' is offloaded, we record the actual offloaded edge node $m_{v'}$ and update $\bar{t_{v'}}$ and $\bar{f_{v'}}$ with Eq. (15) and Eq. (16), respectively. Besides, we update the last offloaded task $l_{m_{v'}}$ as task v' and update the unoffloaded ready set V_R. We repeat this iteration until all tasks have been offloaded. The FS algorithm is formally described in Algorithm 2.

5.2 Performance Analysis

This section analyzes the approximate performance of FS. If we do not consider the services and the number of edge nodes constraints, we can offload tasks satisfying the favorite successor requirement. We first give the approximation ratio for this problem.

Theorem 2. If we do not consider edge node constraints and offload tasks according to the integer solutions obtained by Eq.(18), we can finish all tasks in a makespan at most $\frac{4}{3}$ times of the optimal makespan.

Proof: Let T^{wo} denote the actual makespan and t_v^{wo} denote the actual start time of task $v \in V$ using the integer solutions to offload. Let T^{lp} denote the makespan obtained by linear program, which is the lower-bound of the optimal makespan (denoted as T^{opt}). We denote the weight of link $\langle v, v' \rangle \in E$ as $w\langle v, v' \rangle = e_v + ca_{vv'}y_{vv'}$.

Since we assume the system contains an unlimited number of edge nodes, we can offload any task once it receives all data from successors. Thus, T^{wo} equals to the longest path in the DAG. It means:

$$\begin{aligned} \frac{T^{wo}}{T^{lp}} &\le \max_{\langle v, v' \rangle \in E} \left(\frac{w^{wo}\langle v, v' \rangle}{w^{lp}\langle v, v' \rangle}\right) \\ &= \max_{\langle v, v' \rangle \in E} \left(\frac{e_v + ca_{vv'}\widehat{y}_{vv'}}{e_v + ca_{vv'}\widetilde{y}_{vv'}}\right) \end{aligned} \tag{19}$$

If $\widehat{y}_{vv'} = 0$, then $\frac{e_v + ca_{vv'}\widehat{y}_{vv'}}{e_v + ca_{vv'}\widetilde{y}_{vv'}} \le 1$. Otherwise $\widetilde{y}_{vv'} \ge 0.5$, which means:

$$\frac{e_v + ca_{vv'}\widehat{y}_{vv'}}{e_v + ca_{vv'}\widetilde{y}_{vv'}} \le \frac{e_v + ca_{vv'}}{e_v + 0.5ca_{vv'}} \le \frac{4}{3} \tag{20}$$

The last inequality holds because we assume that $e_v \ge ca_{vv'}$ for this special case. Hence we conclude that:

$$\frac{T^{wo}}{T^{opt}} \le \frac{T^{wo}}{T^{lp}} \le \max_{\langle v, v' \rangle \in E} \left(\frac{e_v + ca_{vv'}\widehat{y}_{vv'}}{e_v + ca_{vv'}\widetilde{y}_{vv'}}\right) \le \frac{4}{3} \tag{21}$$

□

We then give some features of the proposed FS algorithm.

Lemma 3. Let $l' = \min_{v \in V}(|M_v|)$ (*i.e.*, for any task, at least l' edge nodes are configured with required services) and $l'' = l - l'$. We use $T[0, \bar{t_v}]$ to denote the accumulate idle time on

Algorithm 2 FS: Favorite Successor based Algorithm for Homogeneous Scenario

Step 1: Obtaining Favorite Successor
Construct a linear program based on Eq. (18)
Obtain the optimal solution $\widetilde{y}_{vv'}$
Derive an integer solution $\widehat{y}_{vv'}$ for each $\langle v, v' \rangle \in E$
Record the favorite successor (if exists) for each task based on the integer solution
Step 2: Favorite Successor based Offloading
for each task $v \in V$ **do**
  Obtain $Pred(v)$ and $Succ(v)$ according to DAG
  Initialize variables $\bar{t_v}$, $\bar{f_v}$ and $Avail(v)$ to 0
end for
for each edge node $m \in M$ **do**
  Initialize the last offloaded task l_m to none
end for
Compute the set V_R of unoffloaded tasks that all predecessors are offloaded
while $V_R \neq \emptyset$ **do**
  for each task $v \in V_R$ and edge node $m \in M_v$ **do**
    Compute $EST(v, m)$ with Eq. (13)
  end for
  for each edge node $m \in M$ that the last offloaded task l_m have a favorite successor $f_m \in V_R$ **do**
    if $m \in M_{f_m}$ and $EST(f_m, m) < T(f_m, m) + ca_{l_m f_m}$ **then**
      for each task $v \in V_R \bigcap Succ(l_m)$ and $v \neq f_m$ and $m \in M_v$ **do**
        $Avail(v) = \max_{v' \in Pred(v)}(\bar{t_{v'}} + e_{v'} + ca_{v'v})$
        if 1) $Avail(v) \geq EST(f_m, m)$ or 2) there is an edge node $m' \neq m$ such that $EST(v, m') \leq Avail(v)$ and if $m' \in M_{f_{m'}}$ and $EST(f_{m'}, m') < T(f_{m'}, m') + ca_{l_{m'} f_{m'}}$, $v \notin Succ(l_{m'})$ or $v = f_{m'}$ **then**
          Update $EST(v, m) = EST(f_m, m) + e_{f_m}$
        end if
      end for
    end if
  end for
  $EST_{min} = \min_{v \in V_R, m \in M_v} EST(v, m)$
  Choose one task $v' \in V_R$ to offloaded onto edge node $m' \in M_{v'}$ which satisfying $EST(v', m') = EST_{min}$
  Use $m_{v'}$ to record the offloaded edge node
  Update $\bar{t_{v'}}$ and $\bar{f_{v'}}$ with Eq. (15) and Eq. (16), respectively
  Update the last offloaded task $l_{m_{v'}} = v'$
  Update the unoffloaded task set V_R that all predecessors are offloaded
end while

all edge nodes before the actual start time $\bar{t_v}$ of task v. Then we have :

$$T[0, \bar{t_v}] \leq (l' - 1)t_v^{wo} + l''\bar{t_v}$$

Proof: In the worst case, the other l'' edge nodes have not been configured with any service and all tasks require the support of service(s). Thus, all tasks can only be offloaded onto l' edge nodes while leaving other l'' edge nodes idle. That means, the accumulate idle time on l'' edge nodes equals to $l''\bar{t_v}$. If we can show at most $(l' - 1)t_v^{wo}$ accumulate idle time on l' other edge nodes, the proof is finished. This becomes the identical parallel machines scheduling with dependent tasks problem [43], which has been illustrated by previous works such as [43] [44] [45]. □

Lemma 4. Let T_l^{fs} denote the actual makespan by using the FS algorithm. Then we have:

$$T[0, T_l^{fs}] \leq (l' - 1)T^{wo} + l''T_l^{fs}$$

Proof: Let task v be the last completed task by the FS algorithm, by applying Lemma 3, we have

$$\begin{aligned} T[0, T_l^{fs}] &= T[0, \bar{t_v}] + T[\bar{t_v}, T_l^{fs}] \\ &\leq (l' - 1)t_v^{wo} + l''\bar{t_v} + (l - 1)e_v \\ &= (l' - 1)(t_v^{wo} + e_v) + l''(\bar{t_v} + e_v) \\ &\leq (l' - 1)T^{wo} + l''T_l^{fs} \end{aligned} \tag{22}$$

□

Now, we give the approximation performance of our proposed FS algorithm.

Theorem 5. Let T_l^{opt} to denote the optimal makespan for offloading on l edge nodes. We have $\frac{T_l^{fs}}{T_l^{opt}} \leq \frac{l}{l'} + \frac{4}{3}$.

Proof: We know that the accumulated idle time plus the whole tasks execution time is equal to the total time slices. By applying Theorem 2 and Lemma 4, we have:

$$\begin{aligned} lT_l^{fs} &= T[0, T_l^{fs}] + \sum\nolimits_{v \in V} e_v \\ &\leq (l' - 1)T^{wo} + l''T_l^{fs} + \sum\nolimits_{v \in V} e_v \\ \Rightarrow T_l^{fs} &\leq \frac{l' - 1}{l'}T^{wo} + \frac{\sum_{v \in V} e_v}{l'} \\ &\leq \frac{4(l' - 1)}{3l'}T^{opt} + \frac{l}{l'}T_l^{opt} \\ &\leq \frac{4}{3}T_l^{opt} + \frac{l}{l'}T_l^{opt} \end{aligned} \tag{23}$$

□

The penultimate inequality holds because $T_l^{opt} \geq \frac{\sum_{v \in V} e_v}{l}$. Thus, we conclude that FS can achieve an approximate ratio of $\frac{l}{l'} + \frac{4}{3}$, where l is the number of edge nodes and $l' = \min_{v \in V}(|M_v|)$ (*i.e.*, for any task, at least l' edge nodes are configured with required services).

6 Performance Evaluation

This section evaluates the performance of the proposed algorithms by comparing with state-of-the-art methods over multiple application scenarios using real-world applications (from [23]) and data traces (from [24]).

6.1 Performance Metrics and Methodology

We mainly focus on the comparison of makespan in this section, which is one of the most important metrics for the task offloading problem. We compare CP and FS with the following existing approaches.

- The first one is GenDoc [22]. It derives an efficient dynamic programming based algorithm to find the optimal dependent

tasks offloading scheme with fixed service caching. The key characteristic of GenDoc is that one task might be placed and executed on multiple edge nodes repeatedly to avoid communication delay and achieve the objective of makespan minimization. However, this method may consume huge processing resources on edge nodes.

- The second one is the Individual Time Allocation with Greedy Offloading (ITAGS) algorithm [14], which aims at minimizing the communication and computation costs while satisfying makespan constraint. Specifically, ITAGS first uses a binary-relaxed version of the original problem to allocate a completion deadline for each individual task, and then greedily optimizes the offloading of each task subject to its time allowance. For fair comparison, we modify the objective of ITAGS to makespan minimization while satisfying processing resources constraints. In this way, ITAGS can solve the same problem proposed in this paper.
- The third one is the rounding based algorithm, which is illustrated in Section 4.1. We use Rounding to denote this method.
- The last one is the traditional algorithm, denoted as Greedy. This algorithm picks tasks starting from the top of the DAG to keep dependency. Then it offloads each picked task to the edge node with the earliest finish time while satisfying service and resource constraints.

6.2 Simulation Settings

In this section, we introduce the simulation settings, including the generation methods of DAGs, task set settings, and the scenario settings for simulations.

6.2.1 DAG Generation

Similar to [14] [23], we generate DAGs with respect to real-world structures, namely Gaussian Elimination (GE) [46] and Fast Fourier Transform (FFT) [47]. For the GE structure, given the dimension η of a graph, the number of tasks in a GE structure is $\frac{\eta^2+\eta-2}{2}$ [23]. For the FFT structure, we divide this structure into two parts: recursive calls and the butterfly operation. The number of FFT points θ determines the number of tasks in a FFT structure. There are $2 \cdot (\theta - 1) + 1$ recursive call tasks and $\theta \log_2 \theta$ [23]. Both generated structures are well known and used in real-world scenarios. In the following simulations, we generate dependent task set based on the above two structures, denoted by GE Structure and FFT Structure, respectively.

6.2.2 Task Set Settings

Similar to [17] [48], we use the data traces of google clusters [24] to generate task sets. The google cluster track contains hundreds of thousands of jobs (applications). Each job consists of one to thousands of tasks and each task has various parameters specified, including resource requests (*e.g.*, storage and computing requirements) and service requests (*e.g.*, can only be offloaded onto nodes equipped with corresponding services). For other information not included, we use the following methods to generate for our simulations. Specifically, for the general ODT-SC problem, to emulate the heterogeneous environment (*e.g.*, the processing delay of the same job will vary on different edge nodes), we scale the processing time collected from [24] with a factor uniform distribution in (1,10). The communication-to-computation ratio is uniformly randomized in (0.1,10). In other words, for each task pair $\langle v, v' \rangle \in E$, the communication delay for data transmission from task v to task v' is generated through multiplying the processing time for task v with a random number in (0.1,10). Moreover, the required processing resources r_{vm} is drawn uniformly in (1,10) for each task v on edge node m. For each task, the percentage of edge nodes, configured with required services, is denoted by Ω. We set Ω as 50% and the number of edge nodes as 10 by default.

6.2.3 Simulation Scenario Settings

The simulations are performed under two scenarios. Specifically, the first scenario is applied to the heterogeneous environment, *i.e.*, the general ODT-SC problem. We evaluate the performance of CP, GenDoc, ITAGS, Rounding and Greedy under this scenario. The second scenario is applied to the homogeneous environment, *i.e.*, the special case introduced in Section 5. We compare FS with CP, GenDoc, ITAGS, Rounding and Greedy under this scenario.

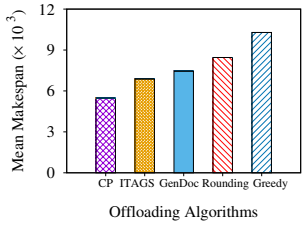
We divide the simulations into six groups and each group of simulations contains the above two scenarios. Basically, we first run 10,000 random test cases to evaluate the overall makespan performance among all algorithms. Then we simulate the mean makespan of these algorithms over a wide range of parameters in the number of tasks, the percentage of edge nodes performing a task, the communication-to-computation ratio and the number of edge nodes.

6.3 Simulation Results

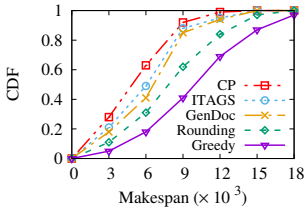
In order to demonstrate the effectiveness of our proposed algorithms, we run six sets of simulations for each DAG structure (*e.g.*, GE and FFT structures) and each simulation scenario (*e.g.*, heterogeneous and homogeneous scenarios).

Overall Performance Comparison: In the first set of simulations, we first randomly generate 200 DAGs using GE and FFT algorithms with the number of tasks from 5 to 500. For each DAG, we generate 50 task set settings according to Section 6.2.2. Thus, we generate totally 10,000 test cases. We perform CP, GenDoc, ITAGS, Rounding and Greedy on these test cases and evaluate the overall makespan performance. The results are shown in Fig. 3. We observe that CP can reduce mean makespan by about 21%, 27%, 35% and 47% compared with ITAGS, GenDoc, Rounding and Greedy, respectively. Besides, as shown in Fig. 3(b), over 60% of test cases will be completed within the makespan of 6000 by CP, while only less than 50% of test cases will be completed within the makespan of 6000 by other algorithms. Thus, in the heterogeneous scenarios, CP achieves better overall makespan performance compared with other benchmarks. That is because our proposed CP algorithm has considered the service caching when making offloading decisions and the algorithm is well-designed (*e.g.*, progressive rounding and computing weights).

Similarly, for each DAG, we generate 50 different random settings that meet the requirements of the homogeneous scenarios. Overall, we generate totally 10,000 test cases for homogeneous

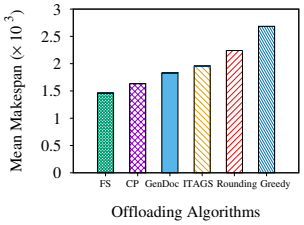


(a) Mean Makespan.

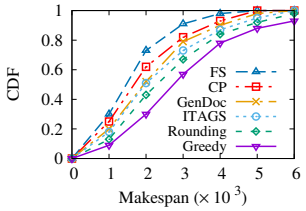


(b) CDF of Makespan.

Fig. 3: Overall Performance for Heterogeneous Scenario.

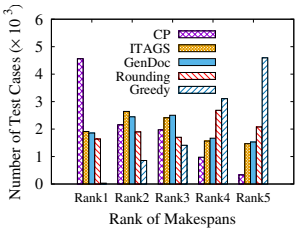


(a) Mean Makespan.

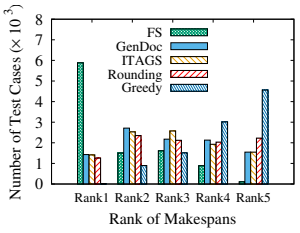


(b) CDF of Makespan.

Fig. 4: Overall Performance for Homogeneous Scenario.



(a) for Heterogeneous Scenario.



(b) for Homogeneous Scenario.

Fig. 5: Number of Test Cases vs. Rank of Makespans.

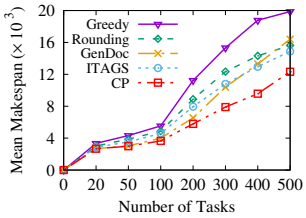
scenarios. Note that, both our proposed CP and FS algorithms can be applied in homogeneous scenarios. Thus, we test FS, CP, GenDoc, ITAGS, Rounding and Greedy on these 10,000 test cases. As shown in Fig. 4, FS reduces the mean makespan by about 11%, 20%, 25%, 35% and 45% compared with CP, GenDoc, ITAGS, Rounding and Greedy, respectively. It means that FS can achieve better makespan results than CP in homogeneous scenarios. That is because CP is specifically designed for homogeneous scenarios and is more efficient than CP in homogeneous scenarios. We usually perform the CP algorithm for heterogeneous scenarios and execute FS for homogeneous scenarios.

Comparison on the Number of Test Cases in Different Rankings: Similar to the first set of simulations, we generate totally 10,000 random heterogeneous test cases and evaluate CP, ITAGS, Rounding and Greedy on these test cases. For each test case, we sort all the algorithms in the ascending order of their makespans. The first algorithm, *i.e.*, with the shortest makespan, is marked as Rank 1 and the algorithm with the second shortest makespan is marked as Rank 2. Similarly, we mark all algorithms for each test case. In the end, we perform 10000 test cases and count the number of times that each algorithm is marked as Rank 1/2/3/4/5. The simulation results are shown in Fig. 5(a). We observe that CP produces the minimum makespan in 45.61% (4,561 out of 10,000) of test cases. By comparison, ITAGS, GenDoc, Rounding and Greedy are in Rank 1 in 1912, 1856, 1639 and 32 test cases, respectively. Our proposed CP algorithm has considered the service caching constraints and adopted well-designed algorithm steps. As a result, CP achieves the shortest makespan in most test cases and produces the longest makespan only in a few test cases. Similarly, we test FS, GenDoc, ITAGS, Rounding and Greedy on 10,000 random homogeneous test cases. As shown in Fig. 5(b), FS produces the shortest makespan in 58.85% (5,885 out of 10,000) of test cases and the longest makespan on very little small portion (less than 1%) of the test cases. By comparison, GenDoc is in Rank 2 for most test cases, ITAGS is in Rank 3 for most cases and Greedy produces the longest makespan for most cases. The results indicate our proposed FS algorithm outperforms other state-of-the-art solutions on most test cases.

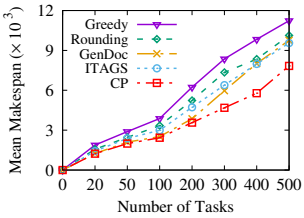
Impact of the Number of Tasks on Makespan: The third set of simulations investigates the mean makespan by changing the number of tasks. We execute each simulation 100 times and average the numerical results. The results are shown in Figs. 6-7. Fig. 6 shows the results for the general ODT-SC problem with different DAG structures. As the number of tasks increases, the mean makespan increases for all algorithms. CP can always achieve lower mean makespan compared with the other four algorithms. For example, when there are 400 tasks in the FFT structure, the mean makespan under CP is 5,783 while 7982, 8010, 8345 and 9834 under ITAGS, GenDoc, Rounding and Greedy, respectively. In other words, CP can decrease mean makespan by about 28%, 29%, 31% and 42% compared with ITAGS, GenDoc, Rounding and Greedy, respectively. Besides, GenDoc is in Rank 2 when the number of tasks is not more than 300 and in Rank 3 when the number of tasks is more than 300. That is because GenDoc may consume more processing resources and encounter resource constraints as the number of tasks increases.

Fig. 7 gives the results for homogeneous scenarios with different DAG structures. We observe that our proposed FS algorithm always outperforms four benchmarks. Basically, FS achieves the shortest mean makespan while CP is in Rank 2, GenDoc is in Rank 3, ITAGS is in Rank 4, Rounding is in Rank 5 and Greedy produces the longest mean makespan. For example, when there are 300 tasks in the GE structure, our proposed FS algorithm can reduce the mean makespan by about 10%, 25%, 30%, 38% and 49% compared with CP, GenDoc, ITAGS, Rounding and Greedy, respectively. That is because FS is specifically designed for homogeneous scenarios and can be more efficient than other algorithms designed for the general ODT-SC problem.

Impact of the Value of Ω on Makespan: The fourth set of simulations shows the mean makespan by changing the value of Ω, *i.e.*, for each task, the percentage of edge nodes that are configured with required services. The results are shown in Figs. 8-9, where the horizontal axes are the value of Ω. As the value of Ω increases, the mean makespan decreases under all algorithms and CP/FS always achieve lower mean makespan than other algorithms. Fig. 8 shows the results for the general ODT-SC

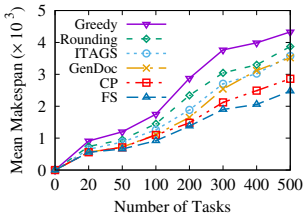


(a) for the GE Structure.

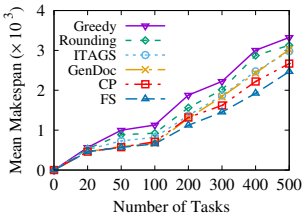


(b) for the FFT Structure.

Fig. 6: Mean Makespan vs. Number of Tasks for Heterogeneous Scenario.

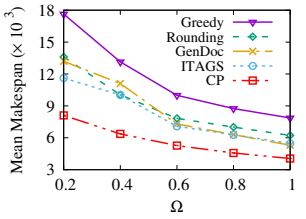


(a) for the GE Structure.

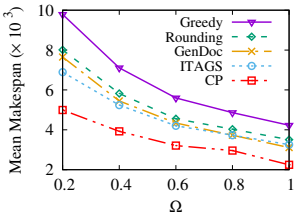


(b) for the FFT Structure.

Fig. 7: Mean Makespan vs. Number of Tasks for Homogeneous Scenario.

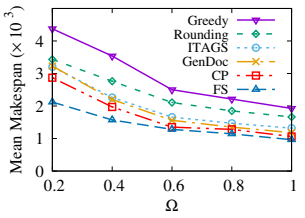


(a) for the GE Structure.

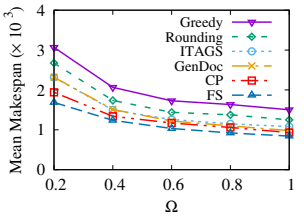


(b) for the FFT Structure.

Fig. 8: Mean Makespan vs. the Value of Ω for Heterogeneous Scenario.

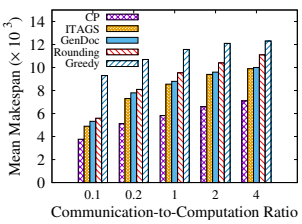


(a) for the GE Structure.

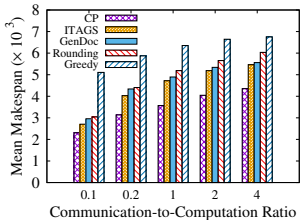


(b) for the FFT Structure.

Fig. 9: Mean Makespan vs. the Value of Ω for Homogeneous Scenario.

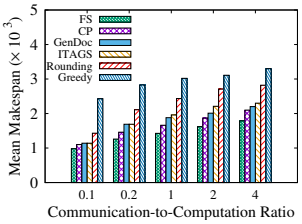


(a) for the GE Structure.

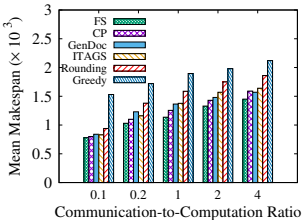


(b) for the FFT Structure.

Fig. 10: Mean Makespan vs. Communication-to-Computation Ratio for Heterogeneous Scenario.



(a) for the GE Structure.



(b) for the FFT Structure.

Fig. 11: Mean Makespan vs. Communication-to-Computation Ratio for Homogeneous Scenario.

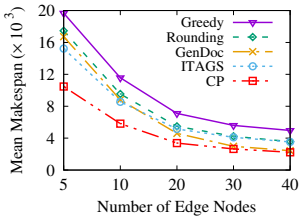
problem with different DAG structures. We observe that CP always achieves the lower makespan compared with the other algorithms. For example, for each task, if we only install required services on 40% of edge nodes in the GE structure, CP will achieve mean makespan of 6368, while ITAGS, Rounding, GenDoc and Greedy can achieve mean makespans of 10024, 10045, 11103 and 13141, respectively. That means CP reduces the mean makespan by about 36.4%, 36.7%, 43% and 51.5% compared with ITAGS, Rounding, GenDoc and Greedy, respectively. When calculating the offloading scheme, CP will take the service caching conditions on each node into account. Thus, compared with other benchmarks, CP will make full use of the limited services on each node and achieve lower makespan.

The results for homogeneous scenarios with different DAG structures are shown in Fig. 9. Regardless of the proportion of the deployed services, FS always achieves lower makespan compared with the other algorithms. For example, when the value of Ω is 0.2 in Fig. 9(b), FS reduces the mean makespan by about 13.1%, 27.1%, 27.1%, 36.9% and 44.9% compared with CP, GenDoc, ITAGS, Rounding and Greedy, respectively. Besides, GenDoc performs better than ITAGS in homogeneous scenarios, especially when the value of Ω increases.

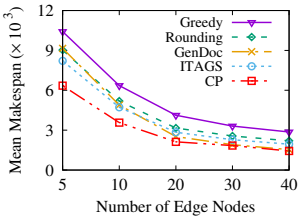
Impact of the Communication-to-Computation Ratio on Makespan: The fifth set of simulations evaluates the mean makespan by changing the communication-to-computation ratio. Specifically, we investigate the impact of inter-nodes communication time on mean makespan. As the communication-to-computation ratio increases, the communication delay will much impact the mean makespan. The results are shown in Figs. 10-11. Fig. 10 presents mean makespan under different communication-to-computation ratio for homogeneous scenarios with different DAG structures and CP always achieves the minimum makespan. For example, when the ratio is 0.2 in the GE structure, CP can reduce the mean makespan by about 29%, 34%, 36% and 52% compared with ITAGS, GenDoc, Rounding and Greedy, respectively.

The impact of the communication-to-computation ratio on makespan for homogeneous scenarios is shown in Fig. 11.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2021.3076687, IEEE Transactions on Parallel and Distributed Systems

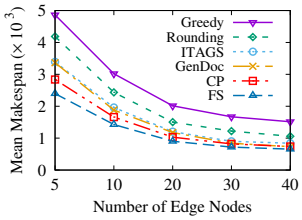


(a) for the GE Structure.

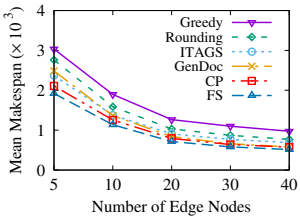


(b) for the FFT Structure.

Fig. 12: Mean Makespan vs. Number of Edge Nodes for Heterogeneous Scenario.



(a) for the GE Structure.



(b) for the FFT Structure.

Fig. 13: Mean Makespan vs. Number of Edge Nodes for Homogeneous Scenario.

FS achieves minimum makespan than other algorithms all the time and GenDoc performs better as the communication-to-computation ratio increases. That is because GenDoc may place one task on multiple edge nodes to avoid the communication overhead.

Impact of the Number of Edge Nodes on Makespan: The last set of simulations illustrates the impact of the number of edge nodes on mean makespan. As shown in Figs. 12-13, with the increasing number of available edge nodes, the mean makespan of all tasks decreases for all algorithms. That is because more edge nodes can provide more computing resources for tasks. For heterogeneous scenarios, CP achieves less mean makespan than the other algorithms. For example, when there are 20 edge nodes with the GE structure, CP reduces the mean makespan by about 26%, 34%, 38% and 52% compared with GenDoc, ITAGS, Rounding and Greedy, respectively. Fig. 13 shows the results for homogeneous scenarios. Regardless of the number of edge nodes in MEC, FS always gets the smaller mean makespan than other algorithms. For example, when there are 10 edge nodes with the FFT structure in MEC, FS can reduce the mean makespan by about 10%, 17%, 18%, 28% and 40% compared with CP, GenDoc, ITAGS, Rounding and Greedy, respectively. By obtaining the favorite successor for each potential node, FS can achieve better makespan performance than other algorithms.

From these simulation results, we can draw some conclusions. First, CP reduces the mean makespan by about 21-47% compared with the other algorithms in heterogeneous scenarios. Second, FS can achieve better performance than CP and reduce mean makespan by about 20-45% compared with the other alternatives in homogeneous scenarios. Third, our proposed CP/FS substantially outperform other algorithms, over a wide range of parameters including the number of tasks, the value of Ω, the communication-to-computation ratio and the number of edge nodes.

6.4 Small-Scale Testbed

Implementation: The prototype system consists of one central controller, four edge nodes, one local device and one remote cloud, as shown in Fig. 14. The central controller, running on a server with a core i9-10900 processor, 64GB RAM, 2TB hard disk and up to 1900Mbps wireless NIC, executes the proposed algorithms to determine the task placement and scheduling scheme. To solve the convex programming problems on the central controller, we

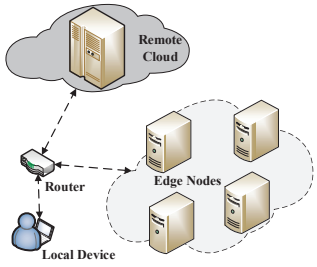


Fig. 14: Prototype System

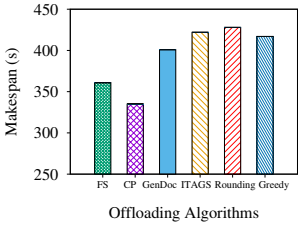


Fig. 15: Makespans of Different Offloading Algorithms

embed the API provided by CPLEX 12.3. The local device, running on a server with a core i5-3470 processor, 8GB RAM, 1TB hard disk and up to 1900Mbps wireless NIC, caches all the task requests. For the four servers who are performing the edge nodes, each of them has an Intel i7-8700 CPU, 16GB RAM, 1TB hard disk and up to 1900Mbps wireless NIC. We use a server equipped with a core i9-10900 processor, 64GB RAM, 2TB hard disk and up to 1900Mbps wireless NIC as the remote cloud. All seven servers are connected through a 3200Mbps router. Note that, in order to estimate the high communication delay between the remote cloud and the local device (or edge nodes), we place the server that acts as the remote cloud far away from the router. Due to the different distance and random noise between these servers and the router, the actual-measured network speed limit between the five servers (acting as edge nodes and the local device) is between 5-13.5 MB/s, while the actual-measured network speed limit between the server acting as the remote cloud and other nodes is between 0.6-2.7 MB/s. To run tasks, we install Hadoop 3.3.0 on our testbed.

Experimental Inputs and Results: We run three Word-Counting applications simultaneously with input size of 1GB, 2GB, and 3GB, respectively on our testbed in Hadoop 3.3.0. Each of the applications consists of several mappers and reducers. Apparently, every reducer can start only after its corresponding mappers complete, which indicates the inherent dependence. In order to simulate the service caching constraints, only 2 edge nodes and the remote cloud can execute mappers.

In the experiment, we run CP, FS, GenDoc, ITAGS, Rounding and Greedy on the testbed and record the makespan performance. The makespans of different algorithms are shown in Fig.

15. We observe that CP can reduce makespan by about 7.5%, 16.4%, 20.5%, 21.8% and 20.7% compared with FS, GenDoc, ITAGS, Rounding and Greedy, respectively. Note that, for the completeness of experiments, we run the FS algorithm in this heterogeneous scenario. The experimental results show that our CP algorithm performs better in the heterogeneous scenario than other algorithms including FS. In addition, due to the small scale of the experiment, the percentages of performance improvement of our algorithms are smaller than the performance improvement in large-scale scenarios. Through the experimental results, we believe that our proposed algorithms can achieve satisfactory performance in real scenarios.

7 Conclusion

In this paper, we have studied the problem of offloading dependent tasks with service caching to minimize the makespan (ODT-SC). We have proved that there exists no constant approximation algorithm for ODT-SC. A convex programming based algorithm has been designed to solve this problem. Moreover, we have studied the special case for the ODT-SC problem (*i.e.*, homogeneous scenario) and proposed an approximate algorithm with bounded approximation factor to solve this practical case. Extensive simulation results have shown the high efficiency of our proposed algorithms.

References

[1] G. Zhao, H. Xu, Y. Zhao, C. Qiao, and L. Huang, "Offloading dependent tasks in mobile edge computing with service caching," in *Proc. IEEE INFOCOM*, 2020, pp. 1–10.

[2] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[3] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 207–215.

[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] T. Ouyang, R. Li, X. Chen, Z. Zhou, and X. Tang, "Adaptive user-managed service placement for mobile edge computing: An online learning approach," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1468–1476.

[6] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[7] M.-H. Chen, B. Liang, and M. Dong, "Joint offloading and resource allocation for computation and communication in mobile cloud with computing access point," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[8] M. Chen and Y. Hao, "Task offloading for mobile edge computing in software defined ultra-dense network," *IEEE Journal on Selected Areas in Communications*, vol. 36, no. 3, pp. 587–597, 2018.

[9] N. Abbas, Y. Zhang, A. Taherkordi, and T. Skeie, "Mobile edge computing: A survey," *IEEE Internet of Things Journal*, vol. 5, no. 1, pp. 450–465, 2017.

[10] Y. Mao, J. Zhang, and K. B. Letaief, "Joint task offloading scheduling and transmit power allocation for mobile-edge computing systems," in *2017 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2017, pp. 1–6.

[11] L. Tong, Y. Li, and W. Gao, "A hierarchical edge cloud architecture for mobile computing," in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[12] S. Bi, L. Huang, and Y.-J. A. Zhang, "Joint optimization of service caching placement and computation offloading in mobile edge computing system," *arXiv preprint arXiv:1906.00711*, 2019.

[13] W. Zhao, R. Chellappa, P. J. Phillips, and A. Rosenfeld, "Face recognition: A literature survey," *ACM computing surveys (CSUR)*, vol. 35, no. 4, pp. 399–458, 2003.

[14] S. Sundar and B. Liang, "Offloading dependent tasks with communication delay and deadline constraint," in *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE, 2018, pp. 37–45.

[15] S. Pasteris, S. Wang, M. Herbster, and T. He, "Service placement with provable guarantees in heterogeneous edge computing systems," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 514–522.

[16] V. Farhadi, F. Mehmeti, T. He, T. La Porta, H. Khamfroush, S. Wang, and K. S. Chan, "Service placement and request scheduling for data-intensive applications in edge clouds," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1279–1287.

[17] N. Eshraghi and B. Liang, "Joint offloading decision and resource allocation with uncertain task computing requirement," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1414–1422.

[18] Z. Meng, H. Xu, L. Huang, P. Xi, and S. Yang, "Achieving energy efficiency through dynamic computing offloading in mobile edge-clouds," in *2018 IEEE 15th International Conference on Mobile Ad Hoc and Sensor Systems (MASS)*. IEEE, 2018, pp. 175–183.

[19] Y.-H. Kao, B. Krishnamachari, M.-R. Ra, and F. Bai, "Hermes: Latency optimal task assignment for resource-constrained mobile computing," *IEEE Transactions on Mobile Computing*, vol. 16, no. 11, pp. 3056–3069, 2017.

[20] Y. Fan, L. Zhai, and H. Wang, "Cost-efficient dependent task offloading for multiusers," *IEEE Access*, vol. 7, pp. 115 843–115 856, 2019.

[21] S. Bi, L. Huang, and Y. J. A. Zhang, "Joint optimization of service caching placement and computation offloading in mobile edge computing systems," *IEEE Transactions on Wireless Communications*, vol. 19, no. 7, pp. 4947–4963, 2020.

[22] L. Liu, H. Tan, S. H.-C. Jiang, Z. Han, X.-Y. Li, and H. Huang, "Dependent task placement and scheduling with function configuration in edge computing," in *Proceedings of the International Symposium on Quality of Service*. ACM, 2019, p. 20.

[23] H. Arabnejad and J. G. Barbosa, "List scheduling algorithm for heterogeneous systems by an optimistic cost table," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 3, pp. 682–694, 2013.

[24] C. Reiss, A. Tumanov, G. R. Ganger, R. H. Katz, and M. A. Kozuch, "Heterogeneity and dynamicity of clouds at scale: Google trace analysis," in *Proceedings of the Third ACM Symposium on Cloud Computing*. ACM, 2012, p. 7.

[25] J. L. D. Neto, S.-Y. Yu, D. F. Macedo, J. M. S. Nogueira, R. Langar, and S. Secci, "Uloof: a user level online offloading framework for mobile edge computing," *IEEE Transactions on Mobile Computing*, vol. 17, no. 11, pp. 2660–2674, 2018.

[26] L. Huang, X. Feng, C. Zhang, L. Qian, and Y. Wu, "Deep reinforcement learning-based joint task offloading and bandwidth allocation for multi-user mobile edge computing," *Digital Communications and Networks*, vol. 5, no. 1, pp. 10–17, 2019.

[27] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "Mobile edge computing: Survey and research outlook," *arXiv preprint arXiv:1701.01090*, 2017.

[28] "Alibaba trace," https://github.com/alibaba/clusterdata.

[29] N. Chen, Y. Yang, T. Zhang, M.-T. Zhou, X. Luo, and J. K. Zao, "Fog as a service technology," *IEEE Communications Magazine*, vol. 56, no. 11, pp. 95–101, 2018.

[30] S. Sardellitti, G. Scutari, and S. Barbarossa, "Joint optimization of radio and computational resources for multicell mobile-edge computing," *IEEE Transactions on Signal and Information Processing over Networks*, vol. 1, no. 2, pp. 89–103, 2015.

[31] Y. Mao, J. Zhang, S. Song, and K. B. Letaief, "Stochastic joint radio and computational resource management for multi-user mobile-edge

computing systems," *IEEE Transactions on Wireless Communications*, vol. 16, no. 9, pp. 5994–6009, 2017.
[32] J. Yan, S. Bi, Y. J. Zhang, and M. Tao, "Optimal task offloading and resource allocation in mobile-edge computing with inter-user task dependency," *IEEE Transactions on Wireless Communications*, vol. 19, no. 1, pp. 235–250, 2019.
[33] M. Dorigo and L. M. Gambardella, "Ant colonies for the travelling salesman problem," *biosystems*, vol. 43, no. 2, pp. 73–81, 1997.
[34] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, Tech. Rep., 1976.
[35] S. Sahni and T. Gonzalez, "P-complete approximation problems," *Journal of the ACM (JACM)*, vol. 23, no. 3, pp. 555–565, 1976.
[36] S. Mitchell, M. OSullivan, and I. Dunning, "Pulp: a linear programming toolkit for python," *The University of Auckland, Auckland, New Zealand*, 2011.
[37] P. Raghavan and C. D. Tompson, "Randomized rounding: a technique for provably good algorithms and algorithmic proofs," *Combinatorica*, vol. 7, no. 4, pp. 365–374, 1987.
[38] I. I. CPLEX, "V12. 1: User's Manual for CPLEX," *International Business Machines Corporation*, vol. 46, no. 53, p. 157, 2009.
[39] Y. Zhao, M. Pithapur, and C. Qiao, "On progressive recovery in interdependent cyber physical systems," in *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2016, pp. 1–6.
[40] T. X. Tran and D. Pompili, "Joint task offloading and resource allocation for multi-server mobile-edge computing networks," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 1, pp. 856–868, 2018.
[41] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 4, pp. 2322–2358, 2017.
[42] E. Ahmed, A. Gani, M. K. Khan, R. Buyya, and S. U. Khan, "Seamless application execution in mobile cloud computing: Motivation, taxonomy, and open challenges," *Journal of Network and Computer Applications*, vol. 52, pp. 154–172, 2015.
[43] C. Hanen and A. Munier, "An approximation algorithm for scheduling dependent tasks on m processors with small communication delays," *Discrete Applied Mathematics*, vol. 108, no. 3, pp. 239–257, 2001.
[44] H. Casanova, A. Legrand, and Y. Robert, *Parallel algorithms*. Chapman and Hall/CRC, 2008.
[45] O. Sinnen, *Task scheduling for parallel systems*. John Wiley & Sons, 2007, vol. 60.
[46] A. K. Amoura, E. Bampis, and J.-C. Konig, "Scheduling algorithms for parallel gaussian elimination with communication costs," *IEEE Transactions on Parallel and Distributed systems*, vol. 9, no. 7, pp. 679–686, 1998.
[47] H. Topcuoglu, S. Hariri, and M.-y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE transactions on parallel and distributed systems*, vol. 13, no. 3, pp. 260–274, 2002.
[48] H. Tan, Z. Han, X.-Y. Li, and F. C. Lau, "Online job dispatching and scheduling in edge-clouds," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.