DP_Greedy: A Two-Phase Caching Algorithm for Mobile Cloud Services

Dong Huang[†], Xiaopeng Fan[†], Yang Wang[†], Shuibing He[‡], Chengzhong Xu[♮]
[†]Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
[‡]College of Computer Science and Technology, Zhejiang University, China
[♮]Information Science, Faculty of Science and Technology,
State Key Laboratory of IoT for Smart City, University of Macau, Macau SAR, China
{dong.huang, xp.fan, yang.wang1}siat.ac.cn, heshuibing@zju.edu.cn, czxu@um.edu.mo

***Abstract*—In this paper, we study the data caching problem in mobile cloud environment where multiple correlated data items could be packed and migrated to serve a predefined sequence of requests. By leveraging the spatial and temporal trajectory of requests, we propose a two-phase caching algorithm. We first investigate the correlation between data items to determine whether or not two data items could be packed to transfer, and then combine an existing dynamic programming (*DP*)-based algorithm and a *greedy* strategy to design a two-phase algorithm, named *DP_Greedy*, for effectively caching these shared data items to serve a predefined sequence of requests. Under homogeneous cost model, we prove the proposed algorithm is at most $2/\alpha$ times worse than the optimal one in terms of the total service cost, where α is the defined *discount factor*, and also show that the algorithm can achieve this results within $O(mn^2)$ time and $O(mn)$ space complexity for m caches to serve a n-length sequence. We evaluate our algorithm by effectively implementing it and comparing it with the non-packing case, the result show the proposed DP_Greedy algorithm not only presents excellent performances but is also more in line with the actual situation.**

***Index Terms*—data caching, data correlations, mobile cloud computing, dynamic programming, greedy strategy, approximation ratio**

I. Introduction

With the rapid development of mobile devices (e.g., smart phone, ipad, PC), mobile cloud computing has received widespread attention and has gradually become a major form of cloud services. It appears in people's daily life more and more frequently, such as remote management, wireless push, storage backup, online search, online music, and mobile notes, etc. A common goal of these services is to maximize the data availability [1], which is also a pressing need for cloud service providers (CSPs) to provide high-quality data services. As mobile cloud services are usually time sensitive, how to reduce the service latency and the network loads to maximize data access efficiency is a crucial issue for cloud computing to offer high-quality data services and lower service costs. However, with the increase of mobile users, the number of visits, the requirement of service quality, and costs of cloud infrastructure rental, achieving these goals is becoming increasingly difficult for cloud service providers. To address these issues, data caching as an effective technology to reduce data service delay and maximize network bandwidth, has been widely studied in the past few decades [2]–[4].

However, when considering data caching as a service in the cloud, things become different from those in classical scenario. Firstly, the data caching strategy in the cloud is often cost-oriented, instead of capacity-oriented as in classical caching problem. This is because that the storage capacity as a resource in the cloud can be viewed as virtually infinite as long as user can afford it. Secondly, the data items are often correlated in accesses, and the accesses are in general trajectory-based, which is also different from the traditional case where the spatial and temporal localities are often exploited for each individual item. This feature comes from the observation that the overwhelming majority of data items are correlative in real world, and thus they are always accessed together. A typical example is a news page where accessing the news text always implies accessing its associated pictures and video clips in the subsequent time. With these distinct features in mind, the cloud based data caching is highly desired to utilize the spatial-temporal trajectory of mobile accesses to the data items while considering the correlationships to minimize the total service cost.

In this paper, we study the caching problem for multiple data items in an off-line form with an assumption that the spatial-temporal trajectory of each data item accesses is available in advance. The rationale behind this assumption is the observation that 93% of human behavior is predictable [5]. In other words, we can predict and exploit the time and place information of requests in advance to some extent.[*] The essence of this study is based on a general observation that packed data items to serve requests jointly are usually more cost effective than transferring each individual one. To this end, we propose a two-phase data caching algorithm, called *DP_Greedy*, to deal with the data caching problem when some data items are correlative in the mobile cloud computing. In the first phase, the algorithm investigates the correlationships among the data items and in the second phase it integrates the dynamic programming (DP)-based optimal off-line data caching algorithm proposed in [6] with a greedy strategy to

[*]How to obtain the predicted sequence of requests is beyond the scope of this paper.

978-1-7281-4734-5/19/$31.00 © 2019 IEEE

handle the caching problem in our model.

Our studies are based on a homogeneous cost model, which means the caching cost per time unit is identical for each storage server and the transfer cost between any pair of servers is also identical. The homogeneous cost model is often studied in literature [6]–[8] since it typically reflects a certain case in reality that the infrastructures for a particular service are often organized as a subset of homogeneous resources, which in turn results in homogeneous computations and communications [9]. On the other hand, for a cloud infrastructure provider (CIP), its billing rates for resources are often fixed across its different data centers (DCs) in a region. [†]

Since the optimal packing problem for given requests in its general form is believed to be NP-hard, the proposed algorithm is an efficient approximated algorithm. We prove the algorithm is at most $2/\alpha$ times worse than the optimal one in terms of the total service cost for a predefined request sequence, where α is the discount factor that is defined to measure the cost saving when two items are packed, compared to non-packing algorithm. Our algorithm can cache the packed data items, probably with multiple copies amongst the m caches, to serve a n-length sequence within $O(mn^2)$ time and $O(mn)$ complexity. We evaluate our algorithm by effectively implementing it and comparing it with the non-packing case, the result show the proposed DP_Greedy algorithm not only presents excellent performances but is also more in line with the actual situation.

Although as a proof of concept, the proposed algorithm only considers to pack two correlative data items, it can be naturally extended to the case where multiple data items could be packed. However, the detection of the correlationships among multiple data items needs much more efforts. The remainder of this paper is organized as follows: In Section II, we survey some related works and compare them with ours. After that, we present our model and the detailed problem formulation in Section III, followed by Section V where we introduce our algorithm based on the cost model. We describe the experiment in Section VI, and conclude the paper in the last section.

II. Related Work

Data caching as a classic technology to reduce communication overhead and lower response delay for data access has been intensively studying in network-based applications for past decades. The earliest studies on the caching problem in network is web caching [11], [12]. Podlipnig et al. [11] conduct a study on the distributed cache replacement strategy, which provides a classification of existing cache technologies. Using this classification, one can describe the advantages and disadvantages of different strategies. Furthermore, it discusses the importance of proxy cache-based replacement strategies and lists some further research topics. In contrast, Wijesundara et al. [12] regard objects replacement problem in the caching strategy as a NP-hard backpack problem. As such, they propose a heuristic-based strategy to optimize object replacements.

[†]For example, the cloud storage pricing as of June 2015 for Google is \$0.026 (GB/Month), for Azure is \$0.030 (GB/Month). The out-network pricing bears a similarity compared to the storage case [10].

TABLE I
Frequently used notation

Symbol	Meaning
μ	uniform caching cost per time unit
λ	uniform transform cost between servers
C_{ij}^{p}	cost from t_i to t_j for data item d_p
α	the discount factor
$\Pi(i)$	the set of all feasible schedule up to r_i
$\phi(n)$	a feasible schedule up to r_n
$\phi^*(n)$	the optimal schedule up to r_n
$A(i,j)$	the correlation matrix between data items
$J(d_i,d_j)$	the Jaccard similarity between d_i and d_j
$r_{p(i)}$	the most recent request in same server for a particular data item
C^*	the optimal cost of our model
C_1^*,C_2^*	the cost of d_1,d_2 in our optimal schedule
C_{1opt},C_{2opt}	the cost of d_1,d_2 in the optimal schedule in [6]
C_{DPG}	the cost of our proposed algorithm
$C_1^{'},C_2^{'}$	the cost to serve requests with only d_1 or d_2 of our proposed algorithm
C_{1G},C_{2G}	the cost of d_1,d_2 using simple greedy algorithm

As an effective technique for reducing network traffic and improving access latency, the cooperative cache [4], [13]–[16] is proposed. Saihan et al. [13] summarize the three main problems in cooperative caching systems, i.e., cache placements, cache replacements, and cache consistency maintenance. Nuggehalli et al. [14] and Tang et al. [15] respectively prove that subject to the constraints of cache consistency and cache capacity, the optimal cache placements is an NP-complete problem, then they both propose approximate algorithms to solve the cache placement problem. Besides, Saleh and Fan present adaptive cooperative caching strategies, respectively [4], [16]. Our problem is different from theirs in terms of system model, cost model, and problem definition since the existing works are all capacity-oriented and aim to maximize the cache hit ratio, and additionally, the cache cost is not taken into consideration. In contrast, our algorithm goal is to minimize the monetary cost under the raised assumption.

The data caching problems can be divided into on-line case and off-line case based on different application scenarios. In the off-line case, the request sequence is known in advance [17], while for the on-line case, we know nothing about the future requests sequence. Veeravalli et al. [7] studied a caching problem for sharing a single data item among a set of fully connected servers in an off-line form, where they obtained an optimal solution within $O(nm^2 \log m)$ time complexity with respect to a homogeneous cost model, here m and n are the number of cache servers and the length of the request sequence, respectively. Later, this result was further improved by Wang et al. [6], who extended the data caching problem to the cloud-based context, and proposed an optimal off-line caching algorithm with $O(mn)$ time and space complexity. Additionally, they also investigated the problem in its on-line form and presented a fast 3-competitive on-line algorithm.

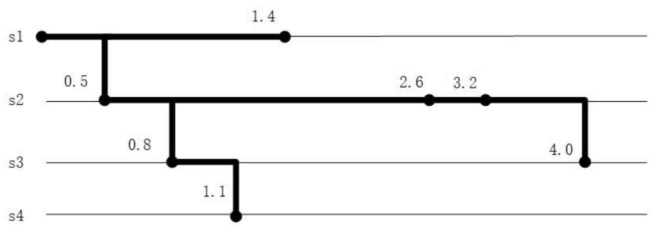

Fig. 1. A feasible schedule (bold lines) with standard form where vertices represent request nodes and the horizontal lines and vertical lines represent cache interval and transfer interval respectively in single data item system. The cost of this schedule is $C = (1.4 + 3.5 + 0.3)\mu + 4\lambda$.

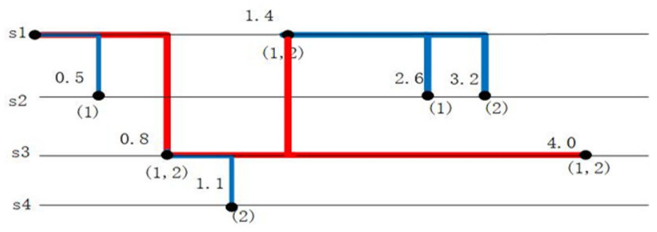

Fig. 2. A feasible schedule based on our system model, the number in bracket represents the data items. Bold lines in red are package migration paths and in blue are paths to serve the requests for only single data item of the package. The cost is $C = ((0.8+3.2)\mu+2\lambda)\times 2\alpha+(0.5+0.3+1.2+1.8)\mu+4\lambda$.

Another similar study is also conducted by Wang et al. [8] where the caching problem for multiple data items is studied with several practical constrains. They proposed an optimal off-line algorithm based on dynamic programming technique and an approximation algorithm within a factor of $1 + C/S$ for a single-copy scenario, where C and S represent the rates for transfer cost and caching cost, respectively.

Our model is very similar with theirs, but it is more realistic and profitable as we incorporate the correlationships between the data items into the algorithm to serve the requests. Our work is derived from the aforementioned work and substantially improved their results in both the problem model and the algorithm efficiency.

III. Problem Formulation

In this section, we describe the data caching problem in details. We first define some useful concepts that will be used in this paper, and then give a standard form of the solution to the problem by following the idea proposed in study [6].

A. System Model

Supposed in a cloud environment, there are k distinct data items with different degrees of correlationships initially stored in a certain server, say s_1. The set of data items is denoted by $D = \{d_1, d_2, ..., d_k\}$, which will be cached in a fully connected network with m cache servers, denoted by $S = \{s_1, s_2, \dots, s_m\}$. A sequence of data requests, $R = \{r_1, r_2, ..., r_n\}$, is made to access these data items, where $r_i =< s_i, t_i, D_i >$ represents the request is made at server s_i at time t_i for a data items subset D_i, $D_i \subseteq D, s_i \in S$. To serve these data requests, the shared data items need to be held in one particular server's cache, transferred from a server to another server, and replicated or destroyed at certain time and server with minimal cost. In off-line settings, we have complete knowledge of where and when each data request is made and which subset of data items are accessed. For simplicity, we also assume that there exists at most one request per time instance as many other previous studies did [6]–[8], so we can use t_i to represent the request node if confuse in not incurred.

Since there are multiple data items in the system, it is likely that several correlated data items could be accessed in one request. In this circumstance, packing these data items as a package to serve the data requests jointly is not only convenient, but also cost effective, and here we use a discount factor α, which is defined to measure the cost saving when multiple items are packed to serve the requests, compared to the case when the packing is not employed, to show these benefits as listed in Table II. In this paper, we only consider two data items package, but without loss of generality, the algorithm is convenient to extend to multiple item case.

We also adopt the space-time diagram as shown in paper [18] to present this problem. Moreover, we continue to use some definitions proposed in [6] that a feasible schedule (shown in Fig. 1, Fig. 2) is that we use cache or transfer to get all requests satisfied along the time line, and a standard form of a schedule is that all transfers occur at the request time and in addition, the paper [7] confirms that there exists at least one optimal schedule which belongs to standard form. What makes our work different from the research [6] is that we consider the multiple data items data caching problem and we take the data items correlationships into consideration as well.

Notably, since a request is often made for a subset of the data items, when we say the request is satisfied often means that the data items in that request mentioned before is satisfied instead of all data items requested for are satisfied.

Definition 1. *For a given data item $d_p (1 \leqslant p \leqslant k)$ in r_i, we use r_{i-1} to represent the most recent request for data item d_p, and $r_{p(i)}$ is the most recent request for data item d_p in the same server.*

B. Cost Model

We adopt the homogeneous cost model in our study, which means the cache cost per time unit for each server is identical, denoted by μ, and the transfer cost between any pair of servers is also identical, denoted by λ. Suppose $C_{ij}^p (1 \leqslant p \leqslant k)$ represents the cost to serve the request $r_j =< s_j, t_j, D_j >$ that contains data item p, $p \in D_j$. And request $r_i =< s_i, t_i, D_i >$, $p \in D_i$, is the most recent request for data item p. We give a formal definition to C_{ij}^p:

$$C_{ij}^p = \begin{cases} (t_j - t_i)\mu + \epsilon\lambda, & t_j > t_i \\ +\infty, & \text{Otherwise} \end{cases} \quad (1)$$

TABLE II
COST OF DATA ITEM SERVING INDIVIDUALLY AND SERVING BY PACKAGE

Data Item	Individual		Package	
	Cache	Transfer	Cache	Transfer
$k = 1$	μ	λ	μ	λ
$k > 1$	$k\mu$	$k\lambda$	$\alpha k\mu$	$\alpha k\lambda$

where ϵ is an $0/1$ variable to signify how to treat the transfer cost in the computation of C^p_{ij}. Specifically, if $s_i = s_j$, then $\epsilon = 0$, which means request r_j is served by a cache from r_i. Otherwise, $\epsilon = 1$, implying data item p is first cached from t_i to t_j, and then transferred from s_i to s_j to serve request r_j. For the data item package, the corresponding cost is $2\alpha C^p_{ij}$, where p is one of packed data item.

C. Problem Goal

In order to satisfy all the data requests, the data items may need to be cached in some server to serve the subsequent requests made on this server or transferred from a server to another so that the requests made on other servers could be satisfied. A transfer operation often implies a replication for a data item, and the copy may be cached in that server and then destroyed in the future for cost saving. Since the replication cost, deletion cost and package cost are always a constant and thus can be added to the transfer cost or the cache cost, without loss of accuracy, we assume these cost are free as many previous studies did [6]–[8].

There are many schedules that can satisfy the requests sequence. Our goal is to find an optimal schedule so that the total cost to serve all these requests is minimum. Here we use $\Pi(i)$ to represent all the feasible schedules to satisfy the requests up to r_i, and $\phi(n)$ and $\phi^*(n)$ denote a feasible schedule and an optimal schedule for this n-length requests sequence, respectively. Each schedule $\phi(n)$ has a cost $cost(\phi(n))$, then the formal definition of our goal is:

$$\phi^*(n) = \underset{\phi(n)\in\Pi(n)}{\arg\min} \quad cost(\phi(n)) \tag{2}$$

in which $cost(\phi(n))$ can be written as:

$$cost(\phi(n)) = \sum_{r_i\in R} cost(r_i) = \sum_{r_i\in R}\sum_{d_j\in D_i} cost(d_j) \tag{3}$$

The packed caching problem in its general form is highly related to the data caching problem under a heterogeneous cost model [7] since the packing operation may change the cost of data migration paths, leading to a variant of the *rectilinear Steiner arborescence problem* [19]. As such, the packed caching problem is believed to be NP-complete. However, its formal proof still remains open. Given the intractability of this problem, we will focus on its approximation solution.

IV. A $2/\alpha$-APPROXIMATION ALGORITHM

In this section, we will first present our two-phase caching algorithm with an approximation factor of $2/\alpha$ based on the problem notation, where α is the discount factor, and then conduct a proof of the approximation ratio.

A. The Algorithm

Suppose we use matrix A to record the correlation between input data items:

$$A(i,j) = \begin{cases} a_{i,j} \in [0,1], & i \neq j \\ 1, & i = j \end{cases} \tag{4}$$

where $i, j \in [0, k-1]$ and it is obvious that the matrix A is a symmetric matrix, meaning $a_{ij} = a_{ji}$. In this paper, we use *Jaccard similarity* (denoted by J), instead of the *co-occurrence* of data items, to specify the correlationships between data items since we expect the DP_Greedy algorithm to perform well when both the frequency and the Jaccard similarity for two data items are high. The Jaccard similarity of two data items is defined as follow:

$$J(d_i, d_j) = \frac{|d_i \cap d_j|}{|d_i \cup d_j|} = \frac{|(d_i, d_j)|}{|d_i| + |d_j| - |(d_i, d_j)} \tag{5}$$

where $|(d_i, d_j)|$ is the number of requests in which data item d_i and d_j co-exist, and $|d_i|$ and $|d_j|$ represent the number of requests that contains data item d_i and the number of requests that access data item d_j, respectively.

We set a correlation threshold denoted by θ, which implies when the correlationships of a pair of data items are greater than the threshold value θ, we pack these two data items to satisfy those requests that simultaneously access to the two data items using the optimal off-line algorithm in [6].

Observation 1. *Since those requests with the two data items in one package are satisfied by the optimal algorithm proposed in [6], which means that there exists at least a single-package copy schedule so that the package is available at any time instance.*

Based on Observation 1, we can reach the conclusion those requests with only one of data item (say d_1 or d_2) of the package, they may be served by a transfer or a cache from the node with that data item or the node with the package, here the node with package only cache or transfer the only requested data item to serve the request. This is because the package is unpacked when serving the front request and therefore we can get any one of data item in the package to serve the subsequent requests. what's more, since the package is available at any time, so we can also directly transfer the package to satisfy those requests. The way we determine to have these requests satisfied depends on if it is most cost effective at this moment.

Observation 2. *By using the greedy algorithm to serve requests r_i with only one of data items in the package, the request may be served by a transfer from r_{i-1} or a cache from $r_{p(i)}$ or a package, and One thing needs to be clarified is that the cost using a package to serve the request with a single data item of the package is always a constant equaling to $2\lambda\alpha$.*

To make it more clear, we take the request 2.6 in Fig. 2 as an example:

- Served by a cache from $r_{p(i)}$ (0.5, r_1), $C^1_{15} = (2.6 - 1.5)\mu$.
- Served by a transfer from r_{i-1} (1.4, r_4), $C^1_{45} = (2.6 - 1.4)\mu + \lambda$.

- Served by a package, $2\lambda\alpha$.

So to serve request r_5, the cost is $\min\{C_{15}^1, C_{45}^1, (2\lambda\alpha)\}$, according to our algorithm. The detailed two phases of this algorithm are shown as follows:

Algorithm 1 The DP_Greedy algorithm

Create an array named *cost* with size $k+1$ to store the cost of each data item, and a dictionary *Jaccard* to store the Jaccard similarity between each data item pair. Create two array *package_list* and *package_flag* of size $k+1$ to respectively store data items packing situation and flag whether the data item is engaged in packing. For easy description, we assume $d_1 = 1, d_2 = 2, ..., d_k = k$.

Inputs: a request sequence with form $r_i =< s_i, t_i, D_i >$
Outputs: the average cost to serve this request sequence denoted by ave_cost

```
1: $\theta \leftarrow \theta_0 \leftarrow 0$;
2: $package\_list[k+1] \leftarrow 0$;
3: $cost[k+1] \leftarrow 0$;
4: $Jaccard \leftarrow 0$;
5: $package\_flag[k+1] \leftarrow 0$;
6: $ave\_cost \leftarrow 0$;
7: **Phase 1** (Jaccard similarity and package)
8: **for** $(i \leftarrow 1\ to\ k-1)$ **do**
9:     **for** $(j \leftarrow i+1\ to\ k)$ **do**
10:        $J(i,j) \leftarrow \frac{|(d_i,d_j)|}{|d_i|+|d_j|-|(d_i,d_j)}$
11:        add ((i,j),$J(i,j)$) to Jaccard
12:    **end for**
13: **end for**
14: sort Jaccard by $J(i,j)$
15: **for** (key in Jaccard) **do**
16:    **if** $(Jaccard(key) > \theta$ and $package\_flag[key.i] \neq 1$
17:        and $package\_flag[key.j] \neq 1)$ **then**
18:        add (key.i, key.j) to package_list
19:        $package\_flag[key.i] \leftarrow 1$;
20:        $package\_flag[key.j] \leftarrow 1$;
21:    **end if**
22: **end for**
23: **for** $(i \leftarrow 1\ to\ k)$ **do**
24:    **if** $(package\_flag[i] = 0)$ **then**
25:        add $d_i$ to package_list
26:    **end if**
27: **end for**
28: **Phase 2** (Cost computation)
29: **for** (item in package_list) **do**
30:    **if** (item size is 1) **then**
31:        **for** $(i \leftarrow 1\ to\ n)$ **do**
32:            **if** (item in $r_i$) **then**
33:                cost[item] += call alg. in [6]
34:            **end if**
35:        **end for**
36:    **else** (item size is 2)
37:        $cost[item.d_1] \leftarrow 0$
38:        **for** $(i \leftarrow 1\ to\ n)$ **do**
39:            **if** $(item.d_1$ and $item.d_2$ in $r_i)$ **then**
40:                $cost[item.d_2]$ += $2\alpha$ (call alg. in [6])
41:            **else if** ( $item.d_1$ or $item.d_2$ in $r_i$) **then**
42:                $cost[item.d_2] += \min\{\mu(t_i - t_{p(i)}),$
43:                    $\lambda + \mu(t_i - t_{i-1}), 2\alpha\lambda\}$
44:            **else**
45:                continue;
46:            **end if**
47:        **end for**
48:    **end if**
49: **end for**
50: $ave\_cost \leftarrow \frac{\sum_{i=1}^{k} cost[i]}{|d_1|+|d_2|+...+|d_k|}$;
51: **return** ave_cost;
```

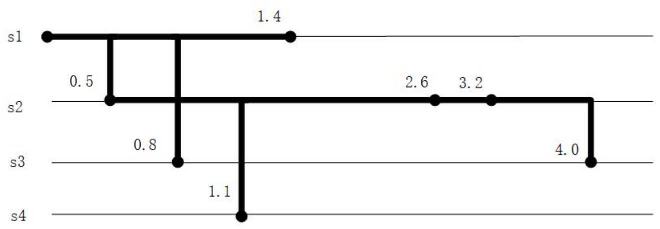

Fig. 3. An example optimal schedule using the optimal off-line algorithm proposed in [6].

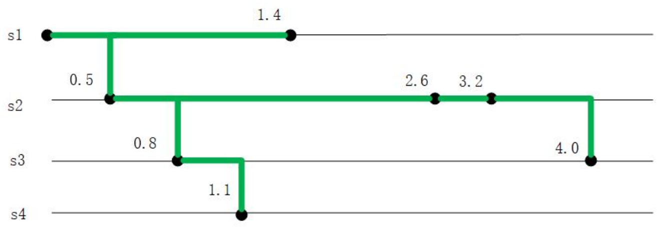

Fig. 4. An example schedule using simple greedy algorithm.

Remarks: Although the proposed algorithm only considers to detect the correlationships between two data items and pack them accordingly, it is not difficult to extend it to the case where multiple correlative data items. However, the correlationships among multiple data items as well as the intractability of this scaling-out (in terms of the number of items) would need much more efforts to detect and deal with.

B. Approximation Ratio Analysis $(2/\alpha)$

In this section, we perform a strict analysis of the approximation ratio of $2/\alpha$. For the sake of easy explanation, we take two data items, say d_1 and d_2, as an example and assume the Jaccard similarity between the two data items is greater than the threshold value, that is the d_1 and d_2 should be packed. Given a sequence of requests, we assume the cost of the optimal algorithm and the DP_Greedy algorithm to satisfy these requests containing data item d_1 or d_2 are C^* and C_{DPG}, respectively. In addition, C_{1opt} and C_{2opt} denote the cost of the optimal off-line algorithm proposed in [6] to serve these requests individually.

Lemma 1. $C^* \geq \alpha(C_{1opt} + C_{2opt})$.

Proof: For the given data requests sequence, there exist an optimal service strategy based on our model that the total cost to serve all these requests is minimal. Data item d_1 and d_2 respectively have a serving tree, the cost is denoted by C_1^* and C_2^* respectively. The set of requests with data item d_1 or d_2 is denoted by *R*, and *R'* represents the set of requests that contains both d_1 and d_2, $R' \subseteq R$, then

$$C_{1opt} \leq C_1^* \quad \text{and} \quad C_{2opt} \leq C_2^*$$

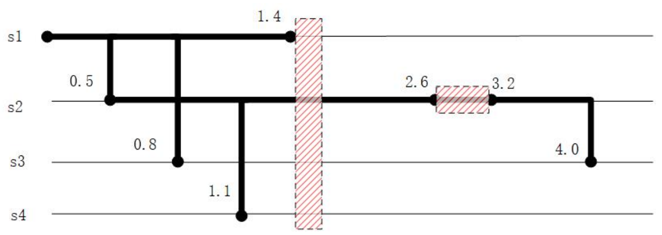

Fig. 5. An example to show how to conduct the cut operation in optimal off-line schedule where the shaded rectangles are the cut regions.

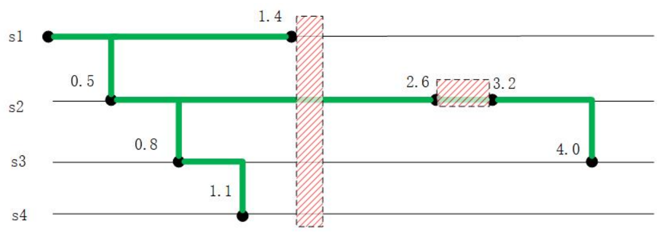

Fig. 6. An example to show how to conduct cut operation in simple greedy schedule in which the shaded rectangles are the cut regions.

$$\begin{aligned}C^* &= \sum_{r_i \in R} C_i = \alpha \sum_{r_j \in R'} C_j + \sum_{r_k \in (R-R')} C_k \\ &\geq \alpha \sum_{r_j \in R'} C_j + \alpha \sum_{r_k \in (R-R')} C_k \\ &= \alpha \sum_{r_j \in R'} (C_{1j} + C_{2j}) + \alpha \sum_{r_k \in (R-R')} C_k \\ &= \alpha(C_1^* + C_2^*) \\ &\geq \alpha(C_{1opt} + C_{2opt})\end{aligned} \tag{6}$$

■

where r_j is the requests with the two packed data items d_1 and d_2, αC_j is the cost of request r_j, and r_k is the request that accesses only one of packed data items and C_k is the corresponding cost. C_{1j} and C_{2j} represents the cost of d_1 and d_2 in r_j (a package), $C_{1j} = C_{2j}$.

$C_{DPG} = C_1' + C_2' + C_{12}$, where C_1' and C_2' are the costs of the greedy strategy to serve the requests with only data item d_1 and the requests with only data item d_2, respectively, and C_{12} is the cost we use dynamic programming technique to satisfy these requests that d_1 and d_2 co-exist. Besides, we use C_{1G} and C_{2G} to denote we individually serve these requests with d_1 or d_2 by the greedy strategy (an example shown in Fig. 4). Then we have following Lemma 2:

Lemma 2. $C_{DPG} \leqslant C_{1G} + C_{2G}$.

Proof: $C_{DPG} = C_1' + C_2' + C_{12}$. For those requests with only d_1 or d_2, the cost of DP_Greedy is not more than the greedy algorithm for the DP_Greedy algorithm has more choice (package). However, to serve those request with d_1 and d_2, the DP_Greedy algorithm adopts the modified optimal algorithm and therefore is better than greedy algorithm, so the cost is less. ■

Now, we first analyze the approximation ratio between the optimal off-line algorithm in [6] and the simple greedy algorithm for a particular data item, and then we deduce the approximation ratio between DP_Greedy and the optimal algorithm. For each situation, we have

1) When $\mu(t_i - t_{p(i)}) \leqslant \lambda$, in this situation, both in the optimal off-line algorithm and the simple greedy algorithm, the request is satisfied by a caching. If we set $\mu = 1$ and $\lambda = 1$, the request 3.2 (Fig. 3 and Fig. 4) meets this requirement.
2) When $\mu(t_i - t_{i-1}) > \lambda$, in this situation, both in the optimal off-line algorithm and the simple greedy algorithm, there exists only one data copy at any time instance during t_{i-1} and t_i. If we set $\mu = 1$ and $\lambda = 1$, the request 2.6 (Fig. 3 and Fig. 4) meet this requirement.
Proof: Assume the request r_i meets this requirement and there exists another data copy between time t_{i-1} and t_i, and we assume this data copy is to serve the request r_k and this data copy derives from r_j. According to our assumption, $t_k > t_i$ and $t_j < t_{i-1}$, the cost to serve r_k is: $C_k = \mu(t_k - t_j) = \mu((t_i - t_j) + (t_k - t_i)) > \mu(t_k - t_i) + \lambda$. However, if we serve r_k by the data copy in r_i, the cost is $C_k = \mu(t_k - t_i) + \lambda$. So that r_k is served by r_i is more profitable than by r_j, the assumption is conflict with the reality, there exist only one data copy between t_{i-1} and t_i when $\mu(t_i - t_{i-1}) > \lambda$. ■
3) For those requests meeting the above two situations, we conduct the following remove operation for both the optimal off-line algorithm and the simple greedy algorithm because they both have the same serving way. The cut operation rules are as follows, in both DP_Greedy and the simple greedy algorithm schedule, for those requests that meet $\mu(t_i - t_{p(i)}) \leqslant \lambda$, the cost of these requests can be ignored, that is the caching line can be removed. While for those requests that meet the requirement $\mu(t_i - t_{i-1}) > \lambda$, we remove the short part of the long caching line (t_i, t_{i-1}) so that $\mu(t_i - t_{i-1})$ equals to λ (shown in Fig. 5 and Fig. 6). In this way, we can obtain the critical state that in the optimal off-line schedule, the cost of a particular request is at least λ, while in the simple greedy schedule, 2λ is the greatest cost for a request. After above processing, the optimal off-line cost for data item d_1 C_{1opt} and d_2 C_{2opt} is changed to C_{1opt}' and C_{2opt}'. Moreover, the cost of the simple greedy algorithm for d_1 C_{1G} and d_2 C_{2G} is denoted by C_{1G}' and C_{2G}', and n' represents the number of requests after above remove operation. So we have

$$\frac{C_{1G}}{C_{1opt}} \leqslant \frac{C_{1G}'}{C_{1opt}'} \leqslant \frac{2n'\lambda}{n'\lambda} = 2 \tag{7}$$

then

$$C_{1G} \leqslant 2C_{1opt} \leqslant 2C_1^* \quad \text{and} \quad C_{2G} \leqslant 2C_{2opt} \leqslant 2C_2^* \tag{8}$$

So far, for data item d_1 and d_2, we get the approximation ratio:

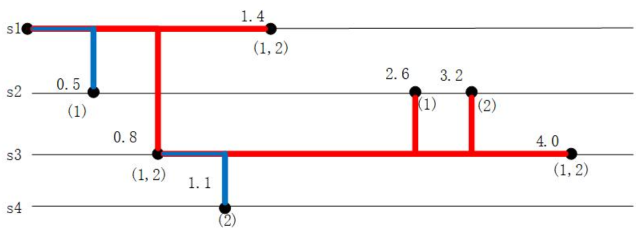

Fig. 7. The schedule derived from our DP_Greedy algorithm for the given requests sequence

$$\frac{C_{DPG}}{C^*} \leqslant \frac{C_1' + C_2' + C_{12}}{\alpha(C_1^* + C_2^*)} \leqslant \frac{C_{1G} + C_{2G}}{\alpha(C_1^* + C_2^*)}$$
$$\leqslant \frac{2(C_1^* + C_2^*)}{\alpha(C_1^* + C_2^*)} = \frac{2}{\alpha}$$

Further, we reach our final conclusion:

$$\frac{\sum_{i=1}^{k} C_{DPG}}{\sum_{i=1}^{k} C^*} \leqslant \frac{C_{DPG}}{C^*} \leqslant \frac{2}{\alpha} \tag{9}$$

Theorem 1. *With homogeneous cost model, the approximation ratio of the* DP_Greedy *algorithm is $2/\alpha$ where α is the discount factor we defined.*

Proof: The proof can be directly given by above discussion. ■

V. Efficient Implementation

In this section, we present the core data structure to implement the proposed algorithm for a given expected sequence of requests, thereby conducting its time and space analysis.

A. Core Data Structure

For efficient implementation of the algorithm, we focus on the second phase of the algorithm by conducting two-pass processing of the sequence of requests, pre-scan pass and service pass, as this phase is the core part of the algorithm. The intent of the pre-scan pass is to build some advanced data structures to represent the requests for efficient processing in the service pass.

1) Pre-Scan Phase: We create the following structures in a pre-scan of the requests. First, for each server, $s^j, 1 \leq j \leq m$, we create a doubly linked list Q_j, which is initialized by a dummy boundary request and used to record the requests made on s_j. Then, for each t_i when a request is made, we maintain a data structure, which contains an m-size pointer array to record the most recent request made on each server relative to t_i. Finally, we also allocate arrays of $A[n]$ and $pLast[m]$ as two global index structures, in which $A[n]$ is designed to index the requests along the time whereas $pLast[m]$ is constantly updated on per-request basis (say, r_i) by storing the immediate request ahead of the request (e.g., r_i) for each server. After updated, $pLast[m]$ is also copied to the m-size pointer array

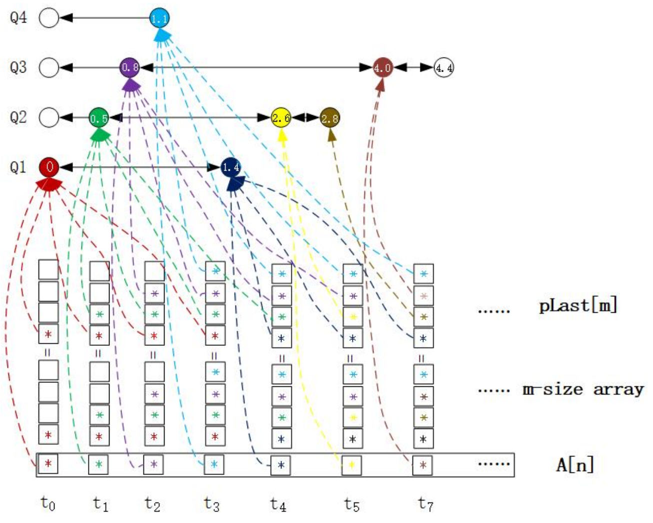

Fig. 8. An example to show how the efficient implementation of the proposed algorithm addresses the data caching problem. The computed cache intervals on each server are also marked, and each pointer, represented by "*" in different colors, is kept up to t_8 (updated when a new request made on that server is processed).

of the request (r_i) so that the m-size pointer array will always keep the most recent requests before or at the request time (say t_i). As such, the most recent request in Q_k relative to r_i in Q_j can be obtained from $pLast[k]$, and these most recent request nodes are potential start nodes of the time intervals that cover r_i.

As r_i is considered, $1 \leq i \leq n$, it is added to the list Q_{s_i} and array $A[i]$, and then $pLast[m]$ is updated and assigned to the request's m-size pointer array, which take $O(m)$ time, there are n request nodes in total and thus take exact $O(mn)$ time and space in pre-scan.

2) Service Phase: During the next service pass over the requests to compute the actual cost, these pointers can be used to precisely identify each of the intervals in $O(mn)$ time per request. By considering the outside loop in Line 31 and 38, this pass totally takes $O(mn^2)$ time.

Fig. 8 is an example to show how the data structures are organized in the efficient implementation of the caching algorithm. During the computation of the recurrences, the algorithm follows the pointer of recent request r_i in $A[i]$ (e.g., $A[7]$) to find the current last element of Q_j (e.g., request node 4.0) and then go back along the backward link of the last element (i.e., node 4.0) to get its previous request node which records $t_{p(i)}$ (e.g., node 0.8). Then, by following the m pointers of the m-size array of node at $t_{p(i)}$ (i.e., node 0.8), each for one server, the required interval on each server can be identified in $O(1)$ (i.e., $\{[0, 1.4], [0.5, 2.6], \varnothing, \varnothing\}$ in our example).

B. Time-Space Complexity Analysis

With the efficient implementation, the time complexity of the algorithm is $O(mn^2)$ while the space remains $O(mn)$.

C. A Running Example

In this example, we take two packed data items as an instance of multiple data items, because once the package strategy is determined, the cost of this data item only relates to the other packed data item. Other data items are either served individually or packed with other data items and has no influence to this two-data items. So taking two packed data items as an example is reasonable.

1) set $\theta = 0.4, \mu = 1, \lambda = 1, \alpha = 0.8$.
2) $J(d_1, d_2) = \frac{|(d_1,d_2)|}{|d_1|+|d_2|-|(d_1,d_2)|} = \frac{3}{7}$.
3) $J(d_1, d_2) > \theta$, so d_1,d_2 are packed to serve those requests with d_1,d_2.
4) Requests(e.g., $0.8, 1.4, 4.0$) are served by the algorithm proposed in [6]:
   - $D(0.8) = +\infty$
     $Tr(0.8) = (0.8 \times 1 + 1) \times 2 \times 0.8 = 2.88$
     $C(0.8) = \min(D(0.8), Tr(0.8)) = 2.88$
   - $D(1.4) = C(0) + (1.4 + 1) \times 2 \times 0.8 = 3.84$
     $Tr(1.4) = C(0.8) + (0.6 + 1) \times 2 \times 0.8 = 5.44$
     $C(1.4) = 3.84$
   - $D(4.0) = \min(C(0.8) + (3.2 + 1) \times 2 \times 0.8, D(1.4) + 3.2 \times 2 \times 0.8) = \min\{9.6, 8.96\} = 8.96$
     $Tr(4.0) = C(1.4) + (2.6 + 1) \times 2 \times 0.8 = 9.6$
     $C(4.0) = 8.96$
5) Those requests with one of data item in a package is served by a greedy algorithm. For d_1 (e.g., $0.5, 2.6$)
   - $D(0.5) = +\infty$
     $Tr(0.5) = C(0) + 0.5 + 1 = 1.5$
     $P = C(0) + 2\alpha\lambda = 1.6$
     $C(0.5) = 1.5$
   - $D(2.6) = C(0.5) + 2.1 = 3.6$
     $Tr(2.6) = C(0.5) + (2.6 - 1.4)\mu + \lambda = 3.7$
     $P = C(0.5) + 2\alpha\lambda = 3.1$
     $C(2.6) = 3.1$
6) For $d_2(e.g., 1.1, 3.2)$:
   - $D(1.1 = +\infty$
     $Tr(1.1) = C(0) + (1.1 - 0.8) \times 1 + 1 = 1.3$
     $P = C(0) + 2\alpha\lambda = 1.6$
     $C(1.1) = 1.3$
   - $D(3.2) = +\infty$
     $Tr(3.2) = C(1.1) + (3.2 - 1.4) \times 1 + 1 = 4.1$
     $P = C(1.1) + 2\alpha\lambda = 2.9$
     $C(2.6) = 2.9$
7) So the total of this schedule is: $8.96 + 3.1 + 2.9 = 14.96$, and the caching schedule is shown in Fig. 7.

VI. Simulation Studies

In this section, we conduct extensive simulation-based experiments to study the performance of our algorithm in reality. We design a solver in C language, which effectively implements our algorithm. The experiment data (shown in Fig. 9) comes from the taxi trace data from Shenzhen, a metropolitan city in South China [20]. We partition the territory of Shenzhen city into a number of parts (e.g., 50), each maintaining a data server to serve the user requests made in the taxis to shared data items.

The algorithm proposed is characterized by several parameters, such as the size of distinct data items k, the number of nodes m in the network, the number of requests n, the discount factor α, the correlation threshold θ, the cache cost μ and transfer cost λ. In order to concentrate our study on the

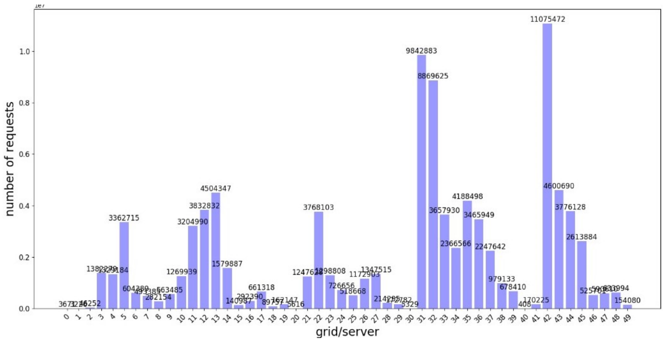

Fig. 9. The distribution of requests in taxi trace data of Shenzhen city.

factors we concerned about, we deliberately ignore some other factors that may make some influence on our algorithm, such as the network traffic, the CPU power and the bandwidth of the network. On the other hand, we take the average cost as the major performance metric since many other performance can be reflected from it such as the network bandwidth occupancy rate.

As for our experiment environment, we randomly select 10 taxis, each accessing a single distinct data item $(d_1, d_2, ..., d_{10})$ as this value can be well handled and without loss of generality to reflect general case. As stated, we partition Shenzhen city into 50 parts, each having a caching server, and select correlation threshold $\theta = 0.3$ and discount factor $\alpha = 0.8$ in this study based on our experience to research human mobility behaviors in metropolitan city [21]. According to the research results [21], the trace of the taxi can be roughly seen as the trace how data items are requested from different servers.

To evaluate our algorithm, we compare the results with those of the optimal off-line algorithm proposed in [6] for a single data item caching since this algorithm has the best results, and can be used as a yardstick to measure the quality of our algorithm.

a) Impact of Jaccard similarity: We first investigate the impact of Jaccard similarity on the proposed algorithm. A bigger value of Jaccard similarity means the ratio of the number of requests with two co-exist packed data items is greater, which undoubtedly is beneficial to our algorithm with respect to the cost to serve these requests. This is what shown in Fig. 11 where the Jaccard similarities come from different pairs of data items in Fig. 10. Since there is no interference between data items except for data items in the same package, we study this impact between packages, for example, 0.5227 is the Jaccard similarity between d_8 and d_9, so the *ave_cost* (y-axis) demonstrates the average cost by using the algorithm proposed in this paper to get all requests with d_8 or d_9 being satisfied, and the rest can be served in the same manner.

As can be seen in the figure, the general trend of this curve is that the bigger Jaccard similarity value of the packed data items is, the better the proposed algorithm performs, and moreover, when the Jaccard similarity is approximately equal to 0.3, the performance of the proposed algorithm and the optimal algorithm for a single data item caching is equally

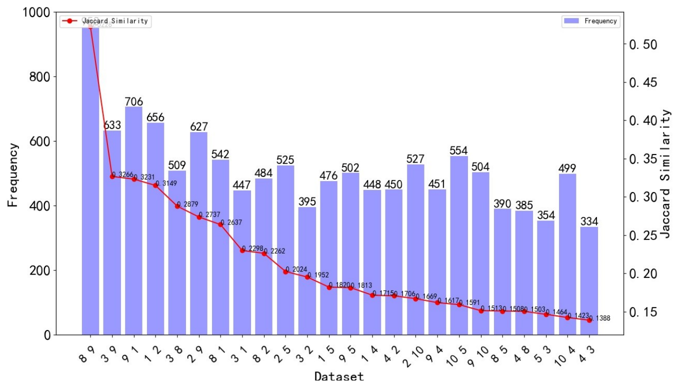
Fig. 10. The frequency and Jaccard similarities in two frequent dataset in taxi trace data.

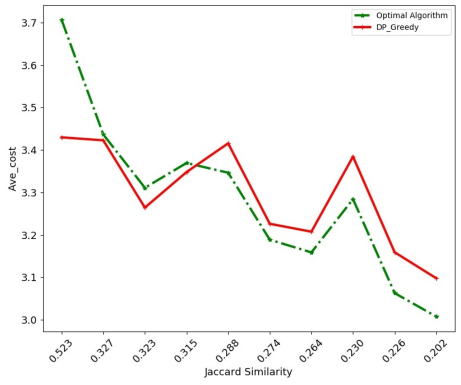
Fig. 11. Impact of Jaccard similarity of two data items on DP_Greedy.

good with respect to this dataset. This is why we set the correlation threshold value to 0.3 in the experiments.

b) Impact of ratio $\rho = \lambda/\mu$: We then consider the impact of ratio $\rho = \lambda/\mu$ on the behavior of the proposed algorithm. To this end, we give the performance of the optimal algorithm for a single data item caching system [6] and compare with that of the proposed algorithm when the ratio ρ is varied from 0.2-5.0, which covers a wide range of this ratio to reflect different cases in reality.

By following the previous arguments, we still set the correlation threshold $\theta = 0.3$ and the discount factor $\alpha = 0.8$ in this experiment. Moreover, to prevent $\lambda + \mu$ from arbitrarily changing and also for a fair comparison, we intentionally limit $\lambda + \mu$ as a constant value of 6. We expect this value in conjunction with their ratio can characterize the relative weights between λ and μ in reality.

Fig. 12 displays the results of our experiment, as shown in the figure, the tendency of average cost exhibits a parabolic like curve that the average cost increases rapidly with the growth of ρ in the initial stage, and then decreases at a slower rate afterwards. This results are highly consistent with our expectation since $\lambda + \mu$ is a constant, and the increase of μ means the decrease of λ and vice versa, which means when ρ is very large or very small (both sides of the curve), the algorithm is able to choose more caching or more transfers (a relatively better way) to satisfy a request for cost saving, so the performance of the algorithm is relatively good.

However, with the increase of ρ, the value of μ and the value of λ gradually reach to equality, under this circumstance, neither the transfer nor the caching is an absolutely favorable choice, which leads to the performance of the algorithm becoming poor. In addition, the first request of each server must be served by a transfer, implying that the transfer cost has greater impact on the algorithm than the caching cost, which explains why the initial stage of this curve is more steeper than its end part and also reveals why the curve reaches its peak around $\rho = 2$ $(\mu = 2, \lambda = 4)$.

c) Impact of discount factor α: It is expected that the benefits from data packing is highly related the discount factor α. By using the same experimental setups, we study the impact

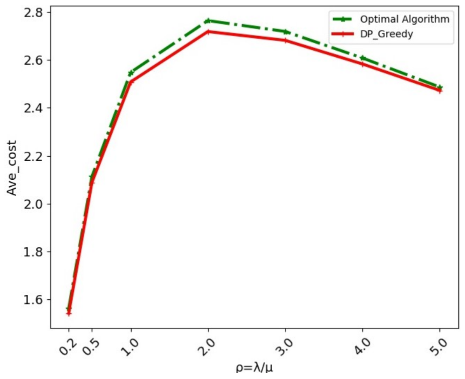
Fig. 12. The relative performance changes of the optimal algorithm and DP_Greedy under different ρs (θ=0.3 and α= 0.8).

of this factor α by varying it from 0.2 to 0.8 to reflect various cases in reality and observe its impact on the service cost by comparing three algorithms *Package_Served*, *Optimal*, and *DP_Greedy*, in which the Package_Served algorithm serves the requests containing data items d_i, d_j, or both by always packing them if they are evaluated to have $J(d_i, d_j)$ greater than the threshold value (i.e. $J(d_i, d_j) \geq \theta$). Package_Served represents the extreme to exploit the data packing from service cost reduction. In contrast, Optimal is on the other extreme where no packing is involved to serve the requests.

Fig. 13 shows the comparison results. It is easy to derive that when α is smaller than 0.5, it is *always* (across all the selected Jaccard similarities) beneficial to pack the correlative data items as shown by Package_Served in Fig. 13 when $\alpha = 0.2$ or 0.4. Also in these cases, Optimal exhibits the worst service cost among the three as it is optimized for single item caching, lacking the power of packing ability to enjoy the benefits of small discount factors. In contrast, the service cost DP_Greedy approaches to that of Package_Served due to its selective packing ability.

As α continues to increase, the performance of the compared algorithms are gradually changed. In particular, the service cost of Optimal consistently declines while that of Package_Served grows up. When $\alpha = 0.8$, DP_Greedy is competitive to Optimal, exhibiting the best performance among the three, especially when $J > 0.3$, while Package_Served is the

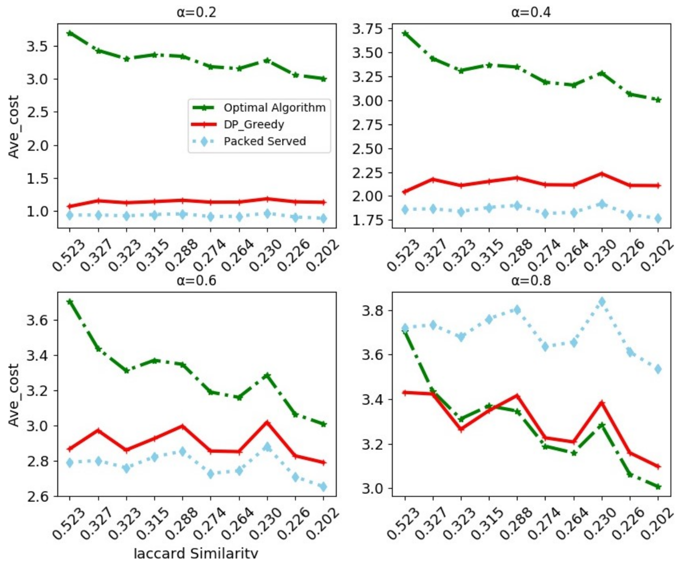

Fig. 13. Impact of discount factor α on average cost

worst due to the value reduction of the data packing.

By observing these results, we can derive that DP_Greedy can effectively exploit the data packing mechanism to optimize the service cost across a range of discount factors and for diverse requests with different Jaccard similarities.

VII. Conclusion

In this paper, we study multiple data items caching problem in the mobile cloud computing system by minimizing monetary cost as our optimization objective. By leveraging the observation that packing two data items to serve requests jointly are usually more cost effective than non-packing, we first investigated the correlation between data items and determine which data items need to be packed together, and then extended the optimal off-line algorithm in [6] with respect to a homogeneous cost model to serve those requests with at most two data items being packed. We proved the proposed algorithm is at most $2/\alpha$ times the optimal algorithm and showed that the result can be achieved within $O(mn^2)$ time and $O(mn)$ space complexity, where α represents the defined discount factor. To evaluate the performance of the algorithm in reality, we implemented it effectively and conducted extensive simulation-based experiments. Our results revealed that the proposed algorithm is not only cost-effective to serve data requests in practice, but also has important theoretical significance to similar problems.

Acknowledgment

This research is partially supported by National Key R&D Program of China (2018YFB1004800) Shenzhen Strategic Emerging Industry Development Funds (JCYJ20170818163026031), Science and Technology Planning Project of Guangdong Province (2019B010137002), National Natural Science Foundation of China (61672513, 61572377), Natural Science Foundation of Hubei Province of China (2017CFC889), Fundamental Research Funds for the Central Universities (2018QNA5015).

References

[1] A. Juels and A. Oprea, “New approaches to security and availability for cloud data,” *Communications of the ACM*, vol. 56, no. 2, pp. 64–73, 2013.

[2] P. Cao and S. Irani, “Cost-aware www proxy caching algorithms.” in *Usenix symposium on internet technologies and systems*, vol. 12, no. 97, 1997, pp. 193–206.

[3] D. Karger, A. Sherman, A. Berkheimer, B. Bogstad, R. Dhanidina, K. Iwamoto, B. Kim, L. Matkins, and Y. Yerushalmi, “Web caching with consistent hashing,” *Computer Networks*, vol. 31, no. 11-16, pp. 1203–1213, 1999.

[4] X. Fan, J. Cao, H. Mao, W. Wu, Y. Zhao, and C. Xu, “Web access patterns enhancing data access performance of cooperative caching in imanets,” in *17th IEEE International Conference on Mobile Data Management (MDM)*, 2016, pp. 50–59.

[5] C. Song, Z. Qu, N. Blumm, and A.-L. Barabási, “Limits of predictability in human mobility,” *Science*, vol. 327, no. 5968, pp. 1018–1021, 2010.

[6] Y. Wang, S. He, X. Fan, C. Xu, J. Culberson, and J. Horton, “Data caching in next generation mobile cloud services, online vs. off-line,” in *Parallel Processing (ICPP), 2017 46th International Conference on*, 2017, pp. 412–421.

[7] B. Veeravalli, “Network caching strategies for a shared data distribution for a predefined service demand sequence,” *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 6, pp. 1487–1497, 2003.

[8] Y. Wang, B. Veeravalli, and C.-K. Tham, “On data staging algorithms for shared data accesses in clouds,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 4, pp. 825–838, 2013.

[9] Y. Wang, B. Veeravalli, and C. K. Tham, “On data staging algorithms for shared data accesses in clouds,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 4, pp. 825–838, April 2013.

[10] Y. Mansouri, A. N. Toosi, and R. Buyya, “Cost optimization for dynamic replication and migration of data in cloud data centers,” *IEEE Transactions on Cloud Computing*, vol. 99, no. 99, pp. 1–1, 2018.

[11] S. Podlipnig and L. Böszörmenyi, “A survey of web cache replacement strategies,” *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[12] M. Wijesundara and T. Tay, “An object replacement strategy for global performance in distributed web caching,” in *Communication Technology Proceedings*, 2003, pp. 1687–1690.

[13] F. Sailhan and V. Issarny, “Cooperative caching in ad hoc networks,” in *International conference on mobile data management*. Springer, 2003, pp. 13–28.

[14] P. Nuggehalli, V. Srinivasan, and C.-F. Chiasserini, “Energy-efficient caching strategies in ad hoc wireless networks,” in *Proceedings of the 4th ACM international symposium on Mobile ad hoc networking & computing*, 2003, pp. 25–34.

[15] B. Tang, H. Gupta, and S. R. Das, “Benefit-based data caching in ad hoc networks,” *IEEE transactions on Mobile Computing*, vol. 7, no. 3, pp. 289–304, 2008.

[16] A. I. Saleh, “An adaptive cooperative caching strategy for mobile ad hoc networks,” *Knowledge-Based Systems*, vol. 120, pp. 133–172, 2017.

[17] P.-R. Lei, T.-J. Shen, W.-C. Peng, and J. Su, “Exploring spatial-temporal trajectory model for location prediction,” in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, vol. 1, 2011, pp. 58–67.

[18] B. Veeravalli and E. M. Yew, “Network caching strategies for reservation-based multimedia services on high-speed networks,” *Data & Knowledge Engineering*, vol. 41, no. 1, pp. 85–103, 2002.

[19] W. Shi and C. Su, “The rectilinear steiner arborescence problem is np-complete,” in *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, ser. SODA ’00, Philadelphia, PA, USA, 2000, pp. 780–787.

[20] S. Cao, Y. Wang, and C. Xu, “Service migrations in the cloud for mobile accesses: A reinforcement learning approach,” in *2017 International Conference on Networking, Architecture, and Storage (NAS)*, 2017, pp. 1–10.

[21] H. Mao, X. Fan, J. Guan, Y.-C. Chen, H. Su, W. Shi, Y. Zhao, Y. Wang, and C. Xu, “Customer attractiveness evaluation and classification of urban commercial centers by crowd intelligence,” *Computers in Human Behavior*, 2018.