Optimal Allocation for Chunked-Reward Advertising

Weihao Kong⋆, Jian Li, Tie-Yan Liu, and Tao Qin

Microsoft Research Asia
Tsinghua University

Abstract. Chunked-reward advertising is commonly used in the industry, such as the guaranteed delivery in display advertising and the daily-deal services (e.g., Groupon) in online shopping. In chunked-reward advertising, the publisher promises to deliver at least a certain volume (a.k.a. tipping point or lower bound) of user traffic to an advertiser according to their mutual contract. At the same time, the advertiser may specify a maximum volume (upper bound) of traffic that he/she would like to pay for according to his/her budget constraint. The objective of the publisher is to design an appropriate mechanism to allocate the user traffic so as to maximize the overall revenue obtained from all such advertisers. In this paper, we perform a formal study on this problem, which we call Chunked-reward Allocation Problem (CAP). In particular, we formulate CAP as a knapsack-like problem with variable-sized items and majorization constraints. Our main results regarding CAP are as follows. (1) We first show that for a special case of CAP, in which the lower bound equals the upper bound for each contract, there is a simple dynamic programming-based algorithm that can find an optimal allocation in pseudo-polynomial time. (2) The general case of CAP is much more difficult than the special case. To solve the problem, we first discover several structural properties of the optimal allocation, and then design a two-layer dynamic programming-based algorithm that can find an optimal allocation in pseudo-polynomial time by leveraging these properties. (3) We convert the two-layer dynamic programming based algorithm to a fully polynomial time approximation scheme (FPTAS), using the technique developed in [8], combined with some careful modifications of the dynamic programs. Besides these results, we also investigate some natural generalizations of CAP, and propose effective algorithms to solve them.

1 Introduction

We study the traffic allocation problem for what we call "chunked-reward advertising". In chunked-reward advertising, an advertiser requests (and pays for) a chunk of advertising opportunities (e.g., user traffic, clicks, or transactions) from a publisher (or ad platform) instead of pursuing each individual advertising opportunity separately (which we call pay-per-opportunity advertising for

⋆ This work is conducted at Microsoft Research Asia.

ease of comparison). More precisely, when an advertiser i submits a request to the publisher, he/she specifies a tuple (l_i, u_i, p_i, b_i), where l_i is the lower bound of advertising opportunities he/she wants to obtain, u_i is the upper bound (which exists mainly due to the budget constraint of the advertiser), p_i is the per-opportunity price he/she is willing to pay, and b_i is a bias term that represents the base payment when the lower bound is achieved. If the number $\mathbf{x}_i$ of opportunities allocated to the advertiser is smaller than l_i, he/she does not need to pay anything because the lower bound is not met; if $\mathbf{x}_i > u_i$, the advertiser only needs to pay for the u_i opportunities and the over allocation is free to him/her. Mathematically, the revenue that the publisher extracts from advertiser i with $\mathbf{x}_i$ opportunities can be written as below:

$$r(\mathbf{x}_i; p_i, l_i, u_i, b_i) = \begin{cases} 0, & \text{if } \mathbf{x}_i < l_i, \\ p_i \mathbf{x}_i + b_i, & \text{if } l_i \leq \mathbf{x}_i \leq u_i, \\ p_i u_i + b_i, & \text{if } u_i < \mathbf{x}_i. \end{cases} \tag{1}$$

1.1 Examples of Chunked-reward Advertising

Many problems in real applications can be formulated as chunked-reward advertising. Below we give two examples: daily-deal services in online shopping and guaranteed delivery in display advertising.

Daily-deal Services In daily-deal services, the publisher (or service provider, e.g., Groupon.com) shows (multiple) selected deals, with significant discount, to Web users every day. The following information of each deal is available for allocating user traffics to these deals.

- The discounted price w_i of the deal. This is the real price with which Web users purchase the deal.
- The tipping point L_i describes the minimum number of purchases that users are required to make in order for the discount of the deal to be invoked; otherwise, the deal fails and no one can get the deal and the discount.
- The purchase limit U_i denotes the maximum number of users that can purchase this deal. The purchase limit is constrained by the service capability of the advertiser/merchant. For example, a restaurant may be able to serve at most 200 customers during the lunch time.
- The revenue share ratio s_i represents the percentage of revenue that the publisher can get from each transaction of the item in the deal. That is, for each purchase of the deal, $w_i s_i$ goes to the publisher and $w_i(1 - s_i)$ goes to the merchant/advertiser. Note that the publisher can get the revenue from a deal i only if the deal is on (i.e., at least L_i purchases are achieved).
- Conversion probability λ_i denotes the likelihood that the i-th deal will be purchased by a web user given that he/she has noticed the deal.

It is straightforward to represent the daily-deal service in the language of chunked-reward advertising, by setting $l_i = L_i/\lambda_i$, $u_i = U_i/\lambda_i$, $p_i = w_i s_i \lambda_i$, and $b_i = 0$, where $\mathbf{x}_i$ is the number of effective impressions[1] allocated to ad i.

Guaranteed Delivery Guaranteed delivery is a major form of display advertising: an advertiser makes a contract with the publisher (or ad platform) to describe his/her campaign goal that the publisher should guarantee. If the publisher can help achieve the campaign goal, it can extract a revenue higher than unguaranteed advertisements. However, if the publisher failed in doing so, a penalization will be imposed.

Specifically, in the contract, the advertiser will specify:

- The number of impressions (denoted by U_i) that he/she wants to achieve;
- The price P_i that he/she is willing to pay for each impression
- The penalty price Q_i that he/she would like to impose on the publisher for each undelivered impression.

If the publisher can successfully deliver U_i impressions for advertiser i, its revenue collected from the advertising is $P_i U_i$; on the other hand, if only $\mathbf{x}_i < U_i$ impressions are delivered, the publisher will be penalized for the undelivered impressions and can only extract a revenue of $\max\{0, P_i U_i - Q_i(U_i - \mathbf{x}_i)\}$.

Again, it is easy to express guaranteed delivery in the language of chunked-reward advertising, by setting $l_i = \frac{Q_i - P_i}{Q_i} U_i$, $u_i = U_i$, $p_i = Q_i$, and $b_i = (P_i - Q_i)u_i$, where $\mathbf{x}_i$ is the number of effective impressions allocated to ad i.

1.2 Chunked-Reward Allocation Problem

A central problem in chunked-reward advertising is how to efficiently allocate the user traffics to the advertisements (deals) so as to maximize the revenue of the publisher (ad platform). For ease of reference, we call such a problem the Chunked Allocation Problem (CAP), which is formulated in details in this subsection.

Suppose that a publisher has M candidate ads to show for a given period of time (e.g., the coming week) and N Web users that will visit its website during the time period[2]. The publisher shows K ads at K slots to each Web user. Without loss of generality, we assume that the i-th slot is better (in terms of attracting the attention of a Web user) than the j-th slot if $i < j$, and use γ_k to denote the discount factor carried by each slot. Similar to the position bias of click probability in search advertising [4, 1], we have that

$$1 \geq \gamma_1 > \gamma_2 > ... > \gamma_K \geq 0.$$

[1] Effective impressions means the real impressions adjusted by the slot discount factor, as shown in the following subsection.

[2] We take the same assumption as in []: the number of visitors can be forecasted and is known to the publisher.

If an ad is shown at slot k to x visitors, we say that the ad has $x\gamma_k$ effective impressions, and use N_k to denote it:

$$N_k = N\gamma_k.$$

We therefore have $N_1 > N_2 > \cdots > N_K$. For simplicity and without much loss of accuracy, we assume that N_k is an integer. We can regard N_k as the expected number of visitors who have paid attention to the k-th slot. With the concept of effective impression, p_i denotes the (expected) revenue that the publisher can get from one effective impression of the i-th ad if the ad is tipped on.

When allocating user traffics to ads, one needs to consider quite a few constraints, which makes CAP a difficult task:

1. Multiple ads will be displayed, one at each slot, for one visitor; one ad cannot be shown at more than one slots for one visitor. This seemingly simple constraints combined with the fact that different slot positions have different discount factors will become challenging to deal with.
2. Each ad has both a lower bound and an upper bound. On one hand, to make money from an ad, the publisher must ensure the ad achieves the lower bound. On the other hand, to maximize revenue, the publisher needs to ensure the traffic allocated to an ad will not go beyond its upper bound. These two opposite forces make the allocation non-trivial.

In the next subsection, we will use formal language to characterize these constraints, and give a mathematical description of the CAP problem.

1.3 Problem Formulation

We use an integer vector $\mathbf{x}$ to denote an allocation, where the i-th element $\mathbf{x}_i$ denotes the number of effective impressions allocated to the i-th ad. For any vector $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$, let $\mathbf{x}_{[1]} \geq \mathbf{x}_{[2]} \geq \ldots \mathbf{x}_{[n]}$ denotes the components of $\mathbf{x}$ in nonincreasing order (ties are broken in an arbitrary but fixed manner). Since there are multiple candidate ads and multiple slots, we need to ensure the feasibility of an allocation. As mentioned before, an allocation $\mathbf{x}$ is *feasible* if it satisfies that (i) no more than one ad is assigned to a slot for any visitor, and (ii) no ad is assigned to more than one slot for any visitor. Actually these constraints are essentially the same in preemptive scheduling of independent tasks on uniform machines. Consider M jobs with processing requirement $\mathbf{x}_i (i = 1, \ldots, M)$ to be processed on K parallel uniform machines with different speeds $N_j (j = 1, \ldots, K)$. Execution of job i on machine j requires $\mathbf{x}_i / N_j$ time units. $\mathbf{x}$ is a feasible allocation if and only if the minimum makespan of the preemptive scheduling problem is smaller or equal to 1. According to [2], the sufficient and necessary conditions for processing all jobs in the interval [0,1] are

$$\frac{\sum_{j=1}^{M} \mathbf{x}_{[j]}}{\sum_{j=1}^{K} N_j} \leq 1, \qquad (2)$$

and

$$\frac{\sum_{j=1}^{i} \mathbf{x}_{[j]}}{\sum_{j=1}^{i} N_j} \leq 1 \text{ , for all } i \leq K. \tag{3}$$

Thus, a vector $\mathbf{x}$ is a feasible allocation for CAP if it satisfies the inequalities in Eqn. (2) and (3).

Based on the above notations, finding an optimal allocation means solving the following optimization problem.

$$\begin{aligned} & \max_{\mathbf{x}} \sum_{i=1}^{M} r(\mathbf{x}_i; p_i, l_i, u_i, b_i) \\ s.t. \quad & l_i \leq \mathbf{x}_i \leq u_i \quad \text{or} \quad \mathbf{x}_i = 0, \text{ for } i = 1, 2, ...M \\ & \mathbf{x} \text{ is a feasible allocation.} \end{aligned}$$

Note that $\mathbf{x}$ is a vector of integers, and we do not explicitly add it as a constraint when the context is clear. The first set of constraints says the number of effective impressions allocated to ad i should be between the lower bound l_i and upper bound u_i.

Note that the feasibility conditions in Eqn. (2) and (3) can be exactly described by the majorization constraints.

Definition 1. ***Majorization constraints***
The vector $\mathbf{x}$ *is* majorized[3] *by vector* $\mathbf{y}$ *(denoted as* $\mathbf{x} \preceq \mathbf{y}$*) if the sum of the largest* i *entries in* $\mathbf{x}$ *is no larger than the sum of the largest* i *entries in* $\mathbf{y}$ *for all* i*, i.e.,*

$$\sum_{j=1}^{i} \mathbf{x}_{[j]} \leq \sum_{j=1}^{i} \mathbf{y}_{[j]}. \tag{4}$$

In the above definition, $\mathbf{x}$ and $\mathbf{y}$ should contain the same number of elements. In Eqn. (2) and (3), N has less elements than $\mathbf{x}$; one can simply add $M - K$ zeros into N (i.e., $N_{[i]} = 0, \forall K < i \leq M$).

Now we are ready to abstract CAP as an combinatorial optimization problem as the following.

Definition 2. ***Problem formulation for* CAP**
There are M *class of items,* $\mathbb{C}_1, \ldots, \mathbb{C}_M$*. Each class* $\mathbb{C}_i$ *is associated with a lower bound* $l_i \in \mathbb{Z}^+$*, an upper bound* $u_i \in \mathbb{Z}^+$*, and a bias term* b_i*. Each item of* $\mathbb{C}_i$ *has a profit* p_i*. We are also given a vector* $\mathbf{N} = \{N_1, N_2, \ldots, N_K\}$*, called the* target vector*, where* $N_1 > N_2 > \cdots > N_K$*. We use* $|\mathbf{N}|$ *to denote* $\sum_{i=1}^{K} N_j$*. Our goal is to choose* $\mathbf{x}_i$ *items from class* $\mathbb{C}_i$ *for each* $i \in [M]$ *such that the following three properties hold:*

[3] In fact, the most rigorous term used here should be "sub-majorize" in mathematics and theoretical computer science literature (see e.g., [10, 6]). Without causing any confusion, we omit the prefix for simplicity.

1. *Either* $\mathbf{x}_i = 0$ *(we do not choose any item of class* $\mathbb{C}_i$ *at all) or* $l_i \leq \mathbf{x}_i \leq u_i$ *(the number of items of class* $\mathbb{C}_i$ *must satisfy both the lower and upper bounds);*
2. *The vector* $\mathbf{x} = \{\mathbf{x}_i\}_i$ *is majorized by the target vector* $\mathbf{N}$ *(i.e.,* $\mathbf{x} \preceq \mathbf{N}$*);*
3. *The total profit of chosen items (adjusted by the class bias term) is maximized.*

1.4 Relation to Scheduling and Knapsack Problems

CAP bears some similarity with the classic parallel machine scheduling problems [11]. The K slots can be viewed as K parallel machines with different speeds (commonly termed as the *uniformly related machines*, see, e.g., [3, 7]). The M ads can be viewed as M jobs. One major difference between CAP and the scheduling problems lies in the objective functions. Most scheduling problems target to minimize some functions related to time given the constraint of finishing all the jobs, such as makespan minimization and total completion time minimization. In contrast, our objective is to maximize the revenue generated from the finished jobs (deals in our problem) given the constraint of limited time.

CAP is similar to the classic knapsack problem in which we want to maximize the total profit of the items that can be packed in a knapsack with a known capacity. In fact, our FPTAS in Section A borrows the technique from [8] for the knapsack problem. Our work is also related to the *interval scheduling* problem [9, 5] in which the goal is to schedule a subset of interval such that the total profit is maximized. CAP differs from these two problems in that the intervals/items (we can think each ad as an interval) have variable sizes.

1.5 Our results

Our major results for CAP can be summarized as follows.

1. (Section 2) As a warmup, we start with a special case of the CAP problem: the lower bound of each class of items equals the upper bound. In this case, we can order the classes by decreasing lower bounds and the order enables us to design a nature dynamic programming-based algorithm which can find an optimal allocation in pseudo-polynomial running time.
2. (Section 3) We then consider the general case of the CAP problem where the lower bound can be smaller than the upper bound. The general case is considerably more difficult than the simple case in that there is no natural order to process the classes. Hence, it is not clear how to extend the previous dynamic program to the general case. To handle this difficulty, we discover several useful structural properties of the optimal allocation. In particular, we can show that the optimal allocation can be decomposed into multiple *blocks*, each of them has at most one *fractional* class (the number of allocated items for the class is less than the upper bound and larger than the lower bound). Moreover, in a block, we can determine for each class except the fractional class, whether the allocated number should be the upper bound

or the lower bound. Hence, within each block, we can reduce the problem to the simpler case where the lower bound of every item equals the upper bound (with slight modifications). We still need a higher level dynamic program to assemble the blocks and need to show that no two different blocks use items from the same class. Our two level dynamic programming-based algorithm can find an optimal allocation in pseudo-polynomial time.

3. (Section A) Using the technique developed in [8], combined with some careful modifications, we can further convert the pseudo-polynomial time dynamic program to a fully polynomial time approximation scheme (FPTAS). We say there is an FPTAS for the problem, if for any fixed constant $\epsilon > 0$, we can find a solution with profit at least $(1-\epsilon)\mathcal{OPT}$ in $\text{poly}(M, K, \log|\mathbf{N}|, 1/\epsilon)$ time (See e.g., [12]).
4. (Appendix B) We consider the generalization from the strict decreasing target vector (i.e., $N_1 > N_2 > \cdots > N_K$) to the non-increasing target vector (i.e., $N_1 \geq N_2 \geq \ldots \geq N_K$), and briefly describe a pseudo-polynomial time dynamic programming-based algorithm for this setting based on the algorithm in Section 3.
5. (Appendix C) For theoretical completeness, we consider for a generalization of CAP where the target vector $\mathbf{N} = \{N_1, \ldots, N_K\}$ may be non-monotone. We provide a $\frac{1}{2} - \epsilon$ factor approximation algorithm for any constant $\epsilon > 0$. In this algorithm, we use somewhat different techniques to handle the majorization constraints, which may be useful in other variants of CAP.

2 Warmup: A Special Case

In this section, we investigate a special case of CAP, in which $l_i = u_i$ for every class. In other words, we either select a fixed number ($\mathbf{x}_i = l_i$) of items from class $\mathbb{C}_i$, or nothing from the class. We present an algorithm that can find the optimal allocation in $\text{poly}(M, K, |\mathbf{N}|)$ time based on dynamic programming.

For simplicity, we assume that the M classes are indexed by the descending order of l_i in this section. That is, we have $l_1 \geq l_2 \geq l_3 \geq \cdots \geq l_M$.

Let $G(i, j, k)$ denote the maximal profit by selecting at most i items from extactly k of the first j classes, which can be expressed by the following integer optimization problem.

$$G(i,j,k) = \max_{\mathbf{x}} \sum_{t=1}^{j} r(\mathbf{x}_t; p_t, l_t, u_t, b_t)$$

$$\text{subject to} \quad \mathbf{x}_t = l_t \quad \text{or} \quad \mathbf{x}_t = 0, \quad \text{for } 1 \leq t \leq j \tag{5}$$

$$\sum_{t=1}^{r} \mathbf{x}_{[t]} \leq \sum_{t=1}^{r} N_t, \quad \text{for } r = 1, 2, ..., \min\{j, K\} \tag{6}$$

$$\sum_{t=1}^{j} \mathbf{x}_{[t]} \leq i \tag{7}$$

$$\mathbf{x}_{[k]} > 0, \quad \mathbf{x}_{[k+1]} = 0 \tag{8}$$

In the above formulation, $\mathbf{x} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_j\}$ is a j dimensional allocation vector. $\mathbf{x}_{[t]}$ is the t-th largest element of vector $\mathbf{x}$. Eqn. (6) restates the majorization constraints. Eqn. (7) ensures that at most i items are selected and Eqn. (8) indicates that exactly k classes of items are selected. Further, we use $Z(i, j, k)$ to denote the optimal allocation vector of the above problem (a j-dimensional vector).

It is easy to see that the optimal profit of the special case is $\max_{1 \le k \le K} G(|\mathbf{N}|, M, k)$. In the following, we present an algorithm to compute the values of $G(i, j, k)$ for all i, j, k.

The Dynamic Program : Initially, we have the base cases that $G(i, j, k) = 0$ if i, j, k all equal zero. For each $1 \le i \le |\mathbf{N}|, 1 \le j \le M, 1 \le k \le j$, the recursion of the dynamic program for $G(i, j, k)$ is as follows.

$$G(i,j,k) = \max \begin{cases} G(i, j-1, k), & \text{if } j > 0 \quad \text{(A)} \\ G(i-1, j, k), & \text{if } i > 0 \quad \text{(B)} \\ G(i - l_j, j-1, k-1) + l_j p_j + b_j, & \text{if } Z(i - l_j, j-1, k-1) \cup l_j \text{ is feasible} \quad \text{(C)} \end{cases} \tag{9}$$

Note that for the case (C) of the above recursion, we need to check whether adding the j-th class in the optimal allocation vector $Z(i - l_j, j - 1, k - 1)$ is feasible, i.e., satisfying the majorization constraints in Eqn. (6). The allocation vector $Z(i, j, k)$ can be easily determined from the recursion as follows.

- If the maximum is achieved at case (A), we have $Z(i, j, k)_t = Z(i, j - 1, k)_t, \forall 1 \le t \le j - 1$, and $Z(i, j, k)_j = 0$.
- If the maximum is achieved at case (B), we have $Z(i, j, k)_t = Z(i-1, j, k)_t, \forall 1 \le t \le j$.
- If the maximum is achieved at case (C), we have $Z(i, j, k)_t = Z(i - l_j, j - 1, k - 1)_t, \forall 1 \le t \le j - 1$, and $Z(i, j, k)_j = l_j$.

According to Eqn. (9), all $G(i, j, k)$ (and thus $Z(i, j, k)$) can be computed in the time[4] of $O(M^2|\mathbf{N}|)$.

At the end of this section, we remark that the correctness of the dynamic program crucially relies on the fact that $u_i = l_i$ for all $\mathbb{C}_i$ and we can process the classes in descending order of their l_is. However, in the general case where $u_i \ne l_i$, we do not have such a natural order to process the classes and the current dynamic program does not work any more.

3 Algorithm for the General CAP

In this section, we consider the general case of CAP ($l_i \le u_i$) and present an algorithm that can find the optimal allocation in poly$(M, K, |\mathbf{N}|)$ time based

[4] One can further decrease the complexity of computing all the $G(i, j, k)$'s to $O(M|\mathbf{N}| \min(M, K))$ by using another recursion equation. We use the recursion equation as shown in Eqn. (9) considering its simplicity for presentation and understanding.

on dynamic programming. Even though the recursion of our dynamic program appears to be fairly simple, its correctness relies on several nontrivial structural properties of the optimal allocation of CAP. We first present these properties in Section 3.1. Then we show the dynamic program in Section 3.2 and prove its correctness. Finally we discuss several extensions of CAP, for which the detailed algorithms are described in Appendix because of space limitations.

3.1 The Structure of the Optimal Solution

Before describing the structure of the optimal solution, we first define some notations.

For simplicity of description, we assume all p_is are distinct [5] and the M classes are indexed in the descending order of p_i. That is, we have that $p_1 > p_2 > \cdots > p_M$. Note that the order of classes in this section is different from that in Section 2.

For any allocation vector $\mathbf{x}$, $\mathbf{x}_i$ indicates the number of items selected from class i, and $\mathbf{x}_{[i]}$ indicates the i-th largest element in vector $\mathbf{x}$. For ease of notions, when we say "class $\mathbf{x}_{[i]}$", we actually refer to the class corresponding to $\mathbf{x}_{[i]}$. In a similar spirit, we slightly abuse the notation $p_{[i]}$ to denote the per-item profit of the class $\mathbf{x}_{[i]}$. For example, $p_{[1]}$ is the per-item profit of the class for which we allocate the most number of items in $\mathbf{x}$ (rather than the largest profit). Note that if $\mathbf{x}_{[i]} = \mathbf{x}_{[i+1]}$, then we put the class with the larger per-item profit before the one with the smaller per-item profit. In other words, if $\mathbf{x}_{[i]} = \mathbf{x}_{[i+1]}$, then we have $p_{[i]} > p_{[i+1]}$.

In an allocation $\mathbf{x}$, we call class $\mathbb{C}_i$ (or $\mathbf{x}_i$) *addable* (w.r.t. $\mathbf{x}$) if $\mathbf{x}_i < u_i$. Similarly, class $\mathbb{C}_i$ (or $\mathbf{x}_i$) is *deductible* (w.r.t. $\mathbf{x}$) if $\mathbf{x}_i > l_i$. A class $\mathbb{C}_i$ is *fractional* if it is both addable and deductible (i.e., $l_i < \mathbf{x}_i < u_i$).

Let $\mathbf{x}^\star$ be the optimal allocation vector. We start with a simple yet very useful lemma.

Lemma 1. *If a deductible class $\mathbb{C}_i$ and an addable class $\mathbb{C}_j$ satisfy $\mathbf{x}^\star_i > \mathbf{x}^\star_j$ in the optimal solution $\mathbf{x}^\star$, we must have $p_i > p_j$ (otherwise, we can get a better solution by setting $\mathbf{x}^\star_i = \mathbf{x}^\star_i - 1$ and $\mathbf{x}^\star_j = \mathbf{x}^\star_j + 1$).*

The proof of lemma is quite straightforward.

The following definition plays an essential role in this section.

Definition 3. *(Breaking Points and Tight Segments) Let the set of* breaking points *for the optimal allocation $\mathbf{x}^\star$ be*

$$P = \{t \mid \sum_{i=1}^{t} \mathbf{x}^\star_{[i]} = \sum_{i=1}^{t} N_i\} = \{t_1 < t_2 < \ldots < t_{|P|}\}.$$

[5] This is without loss of generality. If $p_i = p_j$ for some $i \neq j$, we can break tie by adding an infinitesimal value to p_i, which would not affect the optimality of our algorithm in any way.

To simplify the notations for the boundary cases, we let $t_0 = 0$ *and* $t_{|P|+1} = K$. *We can partition* $\mathbf{x}^\star$ *into* $|P| + 1$ tight segments*,* $S_1, \ldots, S_{|P|+1}$*, where* $S_i = \{\mathbf{x}^\star_{[t_{i-1}+1]}, \mathbf{x}^\star_{[t_{i-1}+2]}, \ldots, \mathbf{x}^\star_{[t_i]}\}$. *We call* $S_{|P|+1}$ *the* tail segment*, and* $S_1, \ldots, S_{|P|}$ non-tail tight segments*.* □

We have the following useful property about the number of items for each class in a non-tail tight segment.

Lemma 2. *Given a non-tail tight segment* $S_k = \{\mathbf{x}^\star_{[t_{k-1}+1]}, \mathbf{x}^\star_{[t_{k-1}+2]}, \ldots, \mathbf{x}^\star_{[t_k]}\}$ *which spans* $N_{t_{k-1}+1}, \ldots, N_{t_k}$. *For each class* $\mathbb{C}_i$ *that appears in* S_k *we must have* $N_{t_{k-1}+1} \geq \mathbf{x}^\star_i \geq N_{t_k}$.

Proof. From the definition $\sum_{i=1}^{t_k} \mathbf{x}^\star_{[i]} = \sum_{i=1}^{t_k} N_i$ and majorization constraint $\sum_{i=1}^{t_k-1} \mathbf{x}^\star_{[i]} \leq \sum_{i=1}^{t_k-1} N_i$ we know that $\mathbf{x}^\star_{[t_k]} \geq N_{t_k}$. As $\mathbf{x}^\star_{[t_k]}$ is the smallest in S_k, we proved $\mathbf{x}^\star_i \geq N_{t_k}$. Similarly form $\sum_{i=1}^{t_{k-1}} \mathbf{x}^\star_{[i]} = \sum_{i=1}^{t_{k-1}} N_i$ and $\sum_{i=1}^{t_{k-1}+1} \mathbf{x}^\star_{[i]} \leq \sum_{i=1}^{t_{k-1}+1} N_i$ we know that $N_{t_{k-1}+1} \geq \mathbf{x}^\star_{[t_{k-1}+1]}$. As $\mathbf{x}^\star_{[t_{k-1}+1]}$ is the biggest in S_k, we proved $N_{t_{k-1}+1} \geq \mathbf{x}^\star_i$. □

Note that as we manually set $t_{|B|+1} = K$, the tail segment actually may not be tight. But we still have $N_{t_{k-1}+1} \geq \mathbf{x}^\star_i$.

Let us observe some simple facts about a tight segment S_k. First, there is at most one fractional class. Otherwise, we can get a better allocation by selecting one more item from the most profitable fractional class and removing one item from the least profitable fractional class. Second, in segment S_k, if $\mathbb{C}_i$ is deductible and $\mathbb{C}_j$ is addable, we must have $p_i > p_j$ (or equivalently $i < j$) . Suppose $\mathbb{C}_{\alpha(S_k)}$ is the per-item least profitable deductible class in S_k and $\mathbb{C}_{\beta(S_k)}$ is the per-item most profitable addable class in S_k. From the above discussion, we know $\alpha(S_k) \leq \beta(S_k)$. If $\alpha(S_k) = \beta(S_k)$, then $\alpha(S_k)$ is the only fractional class in S_k. If there is no deductible class in S_k, we let $\alpha(S_k) = 1$. Similarly, if there is no addable class in S_k, we let $\beta(S_k) = M$. Let us summarize the properties of tight segments in the lemma below.

Lemma 3. *Consider a particular tight segment* S_k *of the optimal allocation* $\mathbf{x}^\star$. *The following properties hold.*

1. *There is at most one fractional class.*
2. *For each class* $\mathbb{C}_i$ *that appears in* S_k *with* $i < \beta(S_k)$*, we must have* $\mathbf{x}^\star_i = u_i$.
3. *For each class* $\mathbb{C}_i$ *that appears in* S_k *with* $i > \alpha(S_k)$*, we must have* $\mathbf{x}^\star_i = l_i$.

Now, we perform the following greedy procedure to produce a coarser partition of $\mathbf{x}^\star$ into disjoint *blocks*, $B_1, B_2, \ldots, B_h$, where each block is the union of several consecutive tight segments. The purpose of this procedure here is to endow one more nice property to the blocks. We overload the definition of $\alpha(B_i)$ ($\beta(B_i)$ resp.) to denote the index of the per-item least (most resp.) profitable deductible (addable resp.) class in B_i. We start with $B_1 = \{S_1\}$. So, $\alpha(B_1) = \alpha(S_1)$ and $\beta(B_1) = \beta(S_1)$. Next we consider S_2. If $[\alpha(B_1), \beta(B_1)]$ intersects with $[\alpha(S_2), \beta(S_2)]$, we let $B_1 \leftarrow B_1 \cup S_2$. Otherwise, we are done with B_1

and start to create B_2 by letting $B_2 = S_2$. Generally, in the i-th step, suppose we are in the process of creating block B_j and proceed to S_i. If $[\alpha(B_j), \beta(B_j)]$ intersects with $[\alpha(S_i), \beta(S_i)]$, we let $B_j \leftarrow B_j \cup S_i$. Note that the new $[\alpha(B_j), \beta(B_j)]$ is the intersection of old $[\alpha(B_j), \beta(B_j)]$ and $[\alpha(S_i), \beta(S_i)]$. Otherwise, we finish creating B_j and let the initial value of B_{j+1} be S_i.

We list the useful properties in the following critical lemma. We can see that Property (2) is new (compared with Lemma 3).

Lemma 4. *Suppose $B_1, \ldots, B_h$ are the blocks created according to the above procedure from the optimal allocation $\mathbf{x}^\star$, and $\alpha(B_i)$ and $\beta(B_i)$ are defined as above. The following properties hold.*

1. *Each block has at most one fractional class.*
2. $\alpha(B_1) \le \beta(B_1) < \alpha(B_2) \le \beta(B_2) < \ldots < \alpha(B_h) \le \beta(B_h)$.
3. *For each class $\mathbb{C}_i$ that appears in any block B_k with $i < \beta(B_k)$, we must have $\mathbf{x}^\star_i = u_i$.*
4. *For each class $\mathbb{C}_i$ that appears in any block B_k with $i > \alpha(B_k)$, we must have $\mathbf{x}^\star_i = l_i$.*

Proof. Consider block $B_k = S_i \cup S_{i+1} \cup \ldots \cup S_j$. It is easy to see from the above procedure that $[\alpha(B_k), \beta(B_k)] = \bigcap_{t=i}^{j} [\alpha(S_t), \beta(S_t)]$. If there are two different fractional class (they must from different S_ts), we have $[\alpha(B_k), \beta(B_k)] = \emptyset$, contradicting the procedure for creating B_k. This proves (1).

Now, we prove (2). Let us first consider two adjacent tight segments S_{i-1} and S_i. By Lemma 1, we have $p_{\alpha(S_{i-1})} > p_{\beta(S_i)}$ (or equivalently, $\alpha(S_{i-1}) < \beta(S_i)$). Suppose we are in the ith step when we are creating block B_j. We can see that $\alpha(B_j) > \beta(S_k)$ for all $k \ge i$. This is because $\alpha(B_j)$ is equal to $\alpha(S_{k'})$ for some $k' < i$. Consider block B_{j+1}. We know it consists of several tight segments S_k with $k \ge i$. So, $\beta(B_{j+1})$ is equal to $\beta(S_k)$ for some $k \ge i$, which is less than $\alpha(B_j)$. Moreover, since intervals $[\alpha(B_j), \beta(B_j)]$ are disjoint, we complete the proof of (2).

Properties (3) and (4) are essentially inherited from Lemma 3. □

3.2 The Dynamic Program

Our algorithm for CAP has two levels, both based on dynamic programming. In the lower level, we attempt to find the optimal allocation for each block. Then in the higher level, we assemble multiple blocks together to form a global optimal solution. Lastly, we prove the optimal allocations for these individual blocks do not use one class of items multiple times, thus can be assembled together.

The Lower Level Dynamic Program : Let us first describe the lower level dynamic program. Denote $F(i, j, k), \forall 1 \le i \le j \le K, 1 \le k \le M$ as the maximal profit generating from the block B which spans $N_i, N_{i+1}, \ldots, N_j$ and $\alpha(B) \le k \le \beta(B)$. Note here the block B is not one of the blocks created from the optimal allocation $\mathbf{x}^\star$, but we still require that it satisfies the properties described in

Lemma 4. More formally, $F(i, j, k)$ can be written as an integer program in the following form:

$$F(i,j,k) = \max \sum_{t=1}^{M} r(\mathbf{x}_t; p_t, l_t, u_t, b_t)$$

$$\text{subject to} \quad \mathbf{x}_t = u_t \quad \text{or} \quad \mathbf{x}_t = 0, \quad \text{for } t < k \tag{10}$$

$$\mathbf{x}_t = l_t \quad \text{or} \quad \mathbf{x}_t = 0, \quad \text{for } t > k \tag{11}$$

$$l_t \le \mathbf{x}_t \le u_t \quad \text{or} \quad \mathbf{x}_t = 0, \quad \text{for } t = k \tag{12}$$

$$\sum_{t=1}^{r} \mathbf{x}_{[t]} \le \sum_{t=i}^{i+r-1} N_t, \quad \text{for } r = 1, 2, ...j - i \tag{13}$$

$$\sum_{t=1}^{j-i+1} \mathbf{x}_{[t]} = \sum_{t=i}^{j} N_t \tag{14}$$

$$\mathbf{x}_{[j-i+2]} = 0. \tag{15}$$

Constraints (10) and (11) correspond to Properties (3) and (4) in Lemma 4. The constraint (12) says $\mathbb{C}_k$ may be the only fractional constraint. The constraints (13) are the majorization constraints. Constraints (14) and (15) say B spans $N_i, \ldots, N_j$ with exactly $j - i + 1$ class of items. If $j = K$ (i.e., it is the last block), we do not have the last two constraints since we may not have to fill all slots, or with fixed number of classes.

To compute the value of $F(i, j, k)$, we can leverage the dynamic program developed in Section 2. The catch is that for any $x_k \in [l_k, u_k]$, according to Eqn. (10) and (11), $\mathbf{x}_i$ can only take 0 or a non-zero value (either u_i or l_i). This is the same as making $u_i = l_i$. Therefore, for a given $x_k \in [l_k, u_k]$, the optimal profit $F(i, j, k)$, denoted as $F_{x_k}(i, j, k)$, can be solved by the dynamic program in Section 2.[6] Finally, we have

$$F(i,j,k) = \max_{x_k = 0, l_k, l_k+1, l_k+2, \cdots, u_k} F_{x_k}(i,j,k).$$

The Higher Level Dynamic Program : We use $D(j, k)$ to denote the optimal allocation of the following subproblem: if $j < K$, we have to fill up exactly $N_1, N_2, \ldots, N_j$ (i.e., $\sum_i \mathbf{x}_i = \sum_{i=1}^{j} N_j$) and $\alpha(B) \le k$ where B is the last block of the allocation; if $j = K$, we only require $\sum_i \mathbf{x}_i \le \sum_{i=1}^{j} N_j$. Note that we still have the majorization constraints and want to maximize the profit. The recursion for computing $D(j, k)$ is as follows:

$$D(j,k) = \max\Big\{ \max_{i<j} \{D(i, k-1) + F(i+1, j, k)\}, D(j, k-1) \Big\}. \tag{16}$$

We return $D(K, M)$ as the final optimal revenue of CAP.

[6] The only extra constraint is (14), which is not hard to ensure at all since the dynamic program in Section 2 also keeps track of the number of slots used so far.

As we can see from the recursion (16), the final value $D(K, M)$ is a sum of several F values, say $F(1, t_1, k_1), F(t_1 + 1, t_2, k_2), F(t_2 + 1, t_3, k_3), \ldots$, where $t_1 < t_2 < t_3 < \ldots$ and $k_1 < k_2 < k_3 < \ldots$. Each such F value corresponds to an optimal allocation of a block. Now, we answer the most critical question concerning the correctness of the dynamic program: whether the optimal allocations of the corresponding blocks together form a global feasible allocation? More specifically, the question is whether one class can appear in two different blocks? We answer this question negatively in the next lemma.

Lemma 5. *Consider the optimal allocations* $\mathbf{x}^1$ *and* $\mathbf{x}^2$ *corresponding to* $F(i_1, j_1, k_1)$ *and* $F(i_2, j_2, k_2)$ *respectively, where* $i_1 \leq j_1 < i_2 \leq j_2$ *and* $k_1 < k_2$. *For any class* $\mathbb{C}_i$, *it is impossible that both* $\mathbf{x}_i^1 \neq 0$ *and* $\mathbf{x}_i^2 \neq 0$ *are true.*

Proof. We distinguish a few cases. We will use Lemma 2 on blocks in the following proof.

1. $i \leq k_1$. Suppose by contradiction that $\mathbf{x}_i^1 \neq 0$ and $\mathbf{x}_i^2 \neq 0$. We always have $\mathbf{x}_i^1 \leq u_i$. Since $i \leq k_1 < k_2$, again by Lemma 4, we have also $\mathbf{x}_i^2 = u_i$. Moreover, from Lemma 2 we know that $\mathbf{x}_i^1 \geq N_{j_1} > N_{i_2} \geq \mathbf{x}_i^2$. This renders a contradiction.
2. $i \geq k_2$. Suppose by contradiction that $\mathbf{x}_i^1 \neq 0$ and $\mathbf{x}_i^2 \neq 0$. By Lemma 4, we know $\mathbf{x}_i^1 = l_i$ and $\mathbf{x}_i^2 \geq l_i$. We also have that $\mathbf{x}_i^1 > N_{i_2} \geq \mathbf{x}_i^2$ due to Lemma 2, which gives a contradiction again.
3. $k_1 < i < k_2$. Suppose by contradiction that $\mathbf{x}_i^1 \neq 0$ and $\mathbf{x}_i^2 \neq 0$. By Lemma 4, we know $\mathbf{x}_i^1 = l_i$ and $\mathbf{x}_i^2 = u_i$. We also have the contradiction by $\mathbf{x}_i^1 > \mathbf{x}_i^2$.

We have exhausted all cases and hence the proof is complete. □

Theorem 1. *The dynamic program* (16) *computes the optimal revenue for* CAP *in time* $\text{poly}(M, K, |\mathbf{N}|)$.

Proof. By Lemma 4, the optimal allocation $\mathbf{x}^\star$ can be decomposed into several blocks $B_1, B_2, \ldots, B_h$ for some h. Suppose B_k spans $N_{i_{k-1}+1}, \ldots, N_{i_k}$. Since the dynamic program computes the optimal value, we have $F(i_{k-1} + 1, i_k, \alpha(B_k)) \geq \sum_{i \in B_k} r(\mathbf{x}_i^\star; p_i, l_i, u_i, b_i)$. Moreover, the higher level dynamic program guarantees that

$$D(K, M) \geq \sum_k F(i_{k-1} + 1, i_k, \alpha(B_k)) \geq \sum_k \sum_{i \in B_k} r(\mathbf{x}_i^\star; p_i, l_i, u_i, b_i) = \mathcal{OPT}.$$

By Lemma 5, our dynamic program returns a feasible allocation. So, it holds that $D(K, M) \leq \mathcal{OPT}$. Hence, we have shown that $D(K, M) = \mathcal{OPT}$. □

3.3 Extensions

We make some further investigations on three extensions for the CAP problem. Due to the space limitation, we only give some high level description in this subsection, and the details are given in Appendix.

First, note the optimal algorithm developed in Section 3.2 runs in pseudo-polynomial time of $|\mathbf{N}|$. If $|\mathbf{N}|$ is very large, one may need some more efficient algorithm. In Part A of Appendix, we present a full polynomial time approximation scheme (FPTAS) for CAP, which can find a solution with profit at least $(1-\epsilon)\mathcal{OPT}$ in time polynomial in the input size (i.e., $O(M + K \times \log|\mathbf{N}|)$) and $1/\epsilon$ for any fixed constant $\epsilon > 0$.

We further consider the general case where $N_1 \geq N_2 \geq \ldots \geq N_K$ and some inequalities hold with equality. Our previous algorithm does not work here since the proof of Lemma 5 relies on strict inequalities. In the general case, the lemma does not hold and we can not guarantee that no class is used in more than one blocks. We refine the algorithm proposed in Section 3.2 and present a new DP algorithm for this general setting in Part B of Appendix.

Third, in Part C of Appendix, we provide a $\frac{1}{2} - \epsilon$ factor approximation algorithm for a generalization of CAP where the target vector $\mathbf{N} = \{N_1, \ldots, N_K\}$ may not be monotone ($N_1 \geq N_2 \geq \ldots \geq N_K$ may not hold). We still require that $\sum_{t=1}^{r} \mathbf{x}_{[t]} \leq \sum_{t=1}^{r} N_t$ for all r. Although we are not aware of an application scenario that would require the full generality, the techniques developed here, which are quite different from those in Section 3.2, may be useful in handling other variants of CAP or problems with similar constraints. So we provide this approximation algorithm for theoretical completeness.

4 Conclusions

We have formulated and studied the traffic allocation problem for chunked-reward advertising, and designed two dynamic programming based algorithms, which can find an optimal allocation in pseudo-polynomial time. An FPTAS has also been derived based on the proposed algorithms, and two generalized settings have been further studied.

There are many research issues related to the CAP problem which need further investigations. (1) We have studied the offline allocation problem and assumed that the traffic N of a publisher is known in advance and all the ads are available before allocation. It is interesting to study the online allocation problem when the traffic is not known in advance and both website visitors and ads arrive online one by one. (2) We have assumed that the position discount γ_i of each slot. It is worthwhile to investigate how to maximize revenue with unknown position discount through online exploration. (3) We have not considered the s-trategic behaviors of advertisers. It is of great interest to study the allocation problem in the setting of auctions and analyze its equilibrium properties.

References

1. G. Aggarwal, A. Goel, and R. Motwani. Truthful auctions for pricing search keywords. In *Proceedings of the 7th ACM conference on Electronic commerce*, pages 1–7. ACM, 2006.
2. P. Brucker. *Scheduling algorithms*, pages 124–125. Springer, fifth edition, 2007.

3. T. Ebenlendr and J. Sgall. Optimal and online preemptive scheduling on uniformly related machines. *STACS 2004*, pages 199–210, 2004.
4. B. Edelman, M. Ostrovsky, and M. Schwarz. Internet advertising and the generalized second price auction: Selling billions of dollars worth of keywords. Technical report, National Bureau of Economic Research, 2005.
5. L. Epstein, L. Jez, J. Sgall, and R. van Stee. Online interval scheduling on uniformly related machines. 2012. Manuscript.
6. A. Goel and A. Meyerson. Simultaneous optimization via approximate majorization for concave profits or convex costs. *Algorithmica*, 44(4):301–323, 2006.
7. E.C. Horvath, S. Lam, and R. Sethi. A level algorithm for preemptive scheduling. *Journal of the ACM (JACM)*, 24(1):32–43, 1977.
8. O.H. Ibarra and C.E. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of the ACM (JACM)*, 22(4):463–468, 1975.
9. R. Lipton. Online interval scheduling. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 302–311. Society for Industrial and Applied Mathematics, 1994.
10. M.A. Nielsen. An introduction to majorization and its applications to quantum mechanics. 2002.
11. M. Pinedo. *Scheduling: theory, algorithms, and systems.* Springer Verlag, 2008.
12. V.V. Vazirani. *Approximation algorithms.* springer, 2004.

Appendix

Because of space limitations, we study three extensions of the chunked-reward advertising problem in this Appendix. To simplify presentations, we assume $b_t = 0, \forall t$ in the following discussions. The proposed algorithms and obtained results can be easily generalized to the setting with nonzero b_t.

A A Full Polynomial Time Approximation Scheme

The optimal algorithm developed in Section 3 runs in pseudo-polynomial time (since it is a polynomial of $|\mathbf{N}|$). In this section, we present a full polynomial time approximation scheme (FPTAS) for CAP. Recall that we say there is an FPTAS for the problem, if for any fixed constant $\epsilon > 0$, we can find a solution with profit at least $(1-\epsilon)\mathcal{OPT}$ in time polynomial in the input size (i.e., $O(M+K\times\log|\mathbf{N}|)$) and $1/\epsilon$. Note that assuming $\mathbf{P} \neq \mathbf{NP}$ this is the best possible approximation algorithm we can obtain since CAP is **NP**-hard (it is a significant generalization of the **NP**-hard knapsack problem).

Our FPTAS is based on the 2-level dynamic programs we developed in Section 3.2. We observe that it is only the lower level dynamic program that runs in pseudo-polynomial time. So our major effort is to convert this dynamic program into an FPTAS. As before, we still need to guarantee at the end that the allocations for these block can be concatenated together to form a global feasible allocation. From now on, we fix the small constant $\epsilon > 0$.

FPTAS for the Lower Level Problem : We would like to approximate the optimal allocation of a block (i.e., $F(i, j, k)$) in polynomial time. Now, we fix i, j and k. Since we can only get an approximate solution, some properties are lost and we have to modify the lower level subproblem (LLS) in the following way.

L1. (10) is strengthened to be $\mathbf{x}_t = \{0, u_t\}$, for $t < k$ and $N_i \geq u_t \geq N_j$;
L2. (11) is strengthened to be $\mathbf{x}_t = \{0, l_t\}$ for $t > k$ and $N_i \geq l_t \geq N_j$;
L3. (12) is strengthened to be $\mathbf{x}_k = 0$ or ($l_k \leq \mathbf{x}_k \leq u_k$ and $N_i \geq \mathbf{x}_k \geq N_j$);
L4. (13) remain the same and (14) is relaxed to $\sum_{t=1}^{j-i+1} \mathbf{x}_{[i]} \leq \sum_{t=i}^{j} N_t$.

Let $F(i, j, k)$ to be the optimal revenue allocation subject to the new set of constraints. From the proof of Lemma 5, we can see the modifications of (10) and (11) do not change the problem the optimal allocation also satisfies the new constraints. (14) is relaxed since we can not keep track of all possible total sizes in polynomial time. The optimal solution of the modified problem is no less than $F(i, j, k)$.

We first assume we know the value of $\mathbf{x}_k$. We will get rid of this assumption later. For ease of description, we need some notations. Let C be the set of classes that may participate in any allocation of LLS (those satisfy L1 and L2). Let s_t be the number of items used in $\mathbb{C}_t$ if $\mathbb{C}_t$ participates the allocation. In other words, $s_t = u_t$ if s_t satisfies L1, $s_t = l_t$ if s_t satisfies L2, and $s_k = \mathbf{x}_k$.

Now, we modify the profit of each class. Let $\tilde{F}$ denote the maximal profit by simply taking one class that satisfies the constraints. It is easy to see that

$$\max_{t \in C} p_t s_t \leq \tilde{F} \leq F(i, j, k).$$

For any class $\mathbb{C}_t$ with $t \neq k$, if $\mathbb{C}_t$ participates the optimal solution of $F(i, j, k)$, we know how many items are used (either u_t or l_t). So, we associate the entire class $\mathbb{C}_t$ with a profit (called *modified class profit*)

$$\tilde{Q}_t = \lfloor \frac{2M p_t s_t}{\epsilon \tilde{F}} \rfloor \text{ for } t \in C$$

The modified profit of $\mathbb{C}_t$ can be seen as a scaled and discretized version of the actual profit of $\mathbb{C}_t$. [7] It is important to note that $\tilde{Q}_t$ is an integer bounded by $O(M/\epsilon)$ and the maximum total modified profit we can allocate is bounded by $O(M^2/\epsilon)$.

Everything is in place to describe the dynamic program. Let $H(t, r, \tilde{Q})$ be the minimum total size of any allocation for subproblem $F(i, j, k)$ with the following set of additional constraints:

1. We can only use classes from $\{\mathbb{C}_1, \ldots, \mathbb{C}_t\}$;
2. Exactly r different classes participate in the allocation;
3. The total modified profit of the allocation is $\tilde{Q}$;
4. All constraints of LLS are also satisfied.

[7] It is critical for us to use the discretized profit as one dimension of the dynamic program instead of the discretized size. Otherwise, we may violate the majorization constraints (by a small fraction). Such idea was also used in the FPTAS for the classic knapsack problem [8].

Initially, $H(t,r,\tilde{Q}) = 0$ for $t,r,\tilde{Q} = 0$ and $H(t,r,\tilde{Q}) = \infty$ for others. The recursion of the dynamic program is as follows:

$$H(t,r,\tilde{Q}) = \min \begin{cases} H(t-1,r,\tilde{Q}), & \text{If we decide } \mathbf{x}_i = 0; \\ H(t-1,r-1,\tilde{Q}-\tilde{Q}_t) + s_t, & \text{If } \mathbb{C}_t \in C \text{ and we use } \mathbb{C}_t \text{ and} \\ & H(t-1,r-1,\tilde{Q}-\tilde{Q}_t) + s_t \le N_i + \ldots + N_{i+r}. \end{cases}$$

The correctness of the recursion is quite straightforward and we omit its proof. We return the allocation $\mathbf{x}$ corresponding to $H(t,r,\tilde{Q})$ that has a finite value and $\tilde{Q}$ is the highest. The running time is bounded by $O((i-j) \times M \times \frac{M^2}{\epsilon})$ which is a polynomial.

Lemma 6. *Suppose* $\mathbf{x}^\star$ *is optimal allocation corresponding to* $F(i,j,k)$ *and we know the value of* $\mathbf{x}^\star_k$. *The profit of the allocation* $\mathbf{x}$ *the above dynamic program is at least* $(1-\epsilon)F(i,j,k)$.

Proof. We use I_t and $I^\star_t$ as the Boolean variables indicating whether $\mathbb{C}_t$ participates in the allocations $\mathbf{x}$ and $\mathbf{x}^\star$ respectively. Since the dynamic program finds the optimal solution with respect to the modified profit, we have

$$\sum_t I_t \tilde{Q}_t \ge \sum_t I^\star_t \tilde{Q}_t.$$

By the definition of the modified profit, we can see that

$$\sum_t I_t \Big(\frac{2Mp_ts_t}{\epsilon\tilde{F}} + 1\Big) \ge \sum_t I_t \Big\lfloor \frac{2Mp_ts_t}{\epsilon\tilde{F}} \Big\rfloor \ge \sum_t I^\star_t \Big\lfloor \frac{2Mp_ts_t}{\epsilon\tilde{F}} \Big\rfloor \ge \sum_t I^\star_t \Big(\frac{2Mp_ts_t}{\epsilon\tilde{F}} - 1\Big).$$

Simple manipulation gives us that

$$\sum_t I_t p_t s_t \ge \sum_t I^\star_t p_t s_t - 2M\frac{\epsilon\tilde{F}}{2M} \ge (1-\epsilon)F(i,j,k).$$

In the last inequality, we use $\tilde{F} \le F(i,j,k)$. This completes the proof.

Lastly, we briefly sketch how we get rid of the assumption that $\mathbf{x}_k$ is known (by losing at most an ϵ fraction of profit). Enumerating all possible $\mathbf{x}_k$ values is not feasible since there are $O(u_k - l_k)$ possibilities in the worst case. To achieve a polynomial running time, we only try the following set of possible values for $\mathbf{x}_k$:

$$D_k = \{0, l_k, u_k, N_i, N_j\} \cup \{\text{all integers in } [l_k,u_k] \cap [N_i,N_j] \text{ with the form} \lfloor (1+\epsilon)^h \rfloor, h \in \mathbb{Z}^+\}.$$

Clearly, the size of D_k is $O(\log|\mathbf{N}|)$. Moreover, for any possible $\mathbf{x}_k$ value, we can see that there is a number in D_k that is at most $\mathbf{x}_k$ and at least $(1-\epsilon)\mathbf{x}_k$. Therefore, for any allocation $\mathbf{x}$, there is an allocation $\tilde{\mathbf{x}}$ where $\tilde{\mathbf{x}}_k \in D_k$ and the profit of $\tilde{\mathbf{x}}$ is at least $1-\epsilon$ times the profit of $\mathbf{x}$.

The Higher Level Problem : The higher level dynamic program is the same as (16), except that we only use the $(1-\epsilon)$-approximation for $F(i,j,k)$. The correctness of the overall algorithm follows the same line as before. Since we have enforced constraints L1, L2 and L3, we can still prove Lemma 5 (all arguments in the proof still carry through). Because we have a $(1-\epsilon)$-approximation for each $F(i,j,k)$ and $D(j,k)$ is a sum of several such F values, we also have a $(1-\epsilon)$-approximation for $D(j,k)$. Moreover, the running time for solving this dynamic program is bounded by a polynomial. In summary, we have the following theorem.

Theorem 2. *There is an FPTAS for* CAP. *In other words, for any fixed constant* $\epsilon > 0$*, we can find a feasible allocation with revenue at least* $(1-\epsilon)\mathcal{OPT}$ *in time* $\text{poly}(M, K, \log|\mathbf{N}|, \frac{1}{\epsilon})$ *where* $\mathcal{OPT}$ *is the optimal revenue.*

B Handling General Nonincreasing Target Vector

In this section, we consider the general case where $N_1 \geq N_2 \geq \ldots \geq N_K$ and some inequalities hold with equality. Our previous algorithm does not work here since the proof of Lemma 5 relies on strict inequalities. In the general case, the lemma does not hold and we can not guarantee that no class is used in more than one blocks. In our experiment, we also found some concrete instances that make the previous algorithm fail. In fact, this slight generalization introduces a lot of complications, especially in the boundary cases. We only briefly sketch some essential changes and omit those tedious (but not difficult) details.

We distinguish two types of blocks, E-blocks and F-blocks. F-blocks are the same as the blocks we defined in Section 3 with one additional constraint: Suppose the F-block spans $N_i, \ldots, N_j$. For any class $\mathbb{C}_k$ in this block, $\mathbf{x}_k > N_j$ if $N_j = N_{j+1}$, and $\mathbf{x}_k < N_i$ if $N_i = N_{i-1}$, To handle the case where there are some consecutive equal N values, we need E-blocks. An E-block consists of a maximal set of consecutive classes, say $\mathbf{x}_{[i]}, \ldots, \mathbf{x}_{[j]}$, such that they are of equal size and form the same number of tight segments. In other words, we have $\sum_{k=1}^{i-1} \mathbf{x}_{[k]} = \sum_{k=1}^{i-1} N_k$ and $\mathbf{x}_{[i]} = \mathbf{x}_{[i+1]} = \ldots = \mathbf{x}_{[j]} = N_i = \ldots = N_j$. An E-block differ from an F-block in that a E-block may contain more than one fractional classes. We can still define $\alpha(E_i)$ and $\beta(E_i)$ for an E-block E_i. But this time we may not have $\alpha(E_i) \leq \beta(E_i)$ due to the presence of multiple fractional classes.

Now, we describe our new dynamic program. Let $E(i,j,k_1,k_2)$ be the optimal allocation for an E-block that spans $N_i, \ldots, N_j$ ($N_i = N_{i+1} = \ldots = N_j$) and $\beta(E_i) \geq k_1$ and $\alpha(E_i) \leq k_2$. Computing $E(i,j,k_1,k_2)$ can be done by a simple greedy algorithm that processes the classes in decreasing order of their profits. We need to redefine the higher level dynamic program. $D(i,k,F)$ represents the optimal allocation for the following subproblem : we have to fill up exactly $N_1, N_2, \ldots, N_i$ (i.e., $\sum_j \mathbf{x}_j = \sum_{j=1}^{i} N_j$) and $\alpha(B) \leq k$ where B is the last block and is an F-block (if $i = K$, we allow $\sum_j \mathbf{x}_j \leq \sum_{j=1}^{i} N_j$). $D(i,k,E)$ represents the optimal allocation of the same subproblem except the last block B is an E-block.

The new recursion is as follows. We first deal with the case where the last block is an F-block.

$$D(i,k,F) = \max_{j<i,l<k} \begin{cases} D(j,l,F) + F(j+1,i,k) \\ D(j,l,E) + F(j+1,i,k) \end{cases} \tag{17}$$

The other case is where the last block is an E-block.

$$D(i,k,E) = \max_{j<i,l\le k} \begin{cases} D(j,l,F) + E(j+1,i,l+1,k) \\ D(j,l,E) + E(j+1,i,l+1,k) \text{ if } N_i \neq N_j \end{cases} \tag{18}$$

We can show that for two consecutive (F or E) blocks B_k and B_{k+1} and in the optimal allocation, we have $\alpha(B_k) < \beta(B_{k+1})$ (even though we may not have $\alpha(B_k) \le \beta(B_k)$ for E-blocks). This is why we set the third argument in $E(j+1,i,l+1,k)$ to be $l+1$. We can also argue that no two blocks would use items from the same class. The proof is similar with Lemma 5. Moreover, we need to be careful about the boundary cases where $\alpha(E)$ and $\beta(E)$ are undefined. In such case, their default values need to be chosen slightly differently. For clarity, we omit those details.

C A $(\frac{1}{2}-\epsilon)$-Approximation When N is Non-monotone

In the section, we provide a $\frac{1}{2}-\epsilon$ factor approximation algorithm for a generalization of CAP where the target vector $\mathbf{N} = \{N_1, \ldots, N_K\}$ may not be monotone ($N_1 \ge N_2 \ge \ldots \ge N_K$ may not hold). We still require that $\sum_{t=1}^{r} \mathbf{x}_{[t]} \le \sum_{t=1}^{r} N_t$ for all r. Although we are not aware of an application scenario that would require the full generality, the techniques developed here, which are quite different from those in Section 3, may be useful in handling other variants of CAP or problems with similar constraints. So we provide this approximation algorithm for theoretical completeness.

Note that our previous algorithms does not work for this generalization. In fact, we even conjecture that the generalized problem is strongly NP-hard (So it is unlike to have a pseudo-polynomial time algorithm). Next, we present our algorithm assuming $|\mathbf{N}|$ is polynomially bounded. At the end, we discuss how to remove this assumption briefly.

We first transform the given instance to a simplified instance. The new instance enjoys a few extra nice properties which make it more amenable to the dynamic programming technique. In the next section, we present a dynamic program for the simplified instance that runs in time poly$(M, K, |\mathbf{N}|)$ for any fixed constant $0 < \epsilon < \frac{1}{3}$ (ϵ is the error parameter).

C.1 Some Simplifications of The Instance

Let $\mathcal{OPT}$ be the optimal value of the original problem. We make the following simplifications. We first can assume that $u_i/l_i \le 1/\epsilon$, for every class $\mathbb{C}_i$. Otherwise, we can replace $\mathbb{C}_i$ by a collection of classs $\mathbb{C}_{i1}, \mathbb{C}_{i2}, \ldots, \mathbb{C}_{ik}$ where

$l_{i1} = l_i, u_{i1} = l_i/\epsilon$, $l_{i2} = l_i/\epsilon + 1, u_{i2} = l_i/\epsilon^2$, $l_{i3} = l_i/\epsilon^2 + 1, \ldots$, $l_{ik} = l_i/\epsilon^{k-1} + 1, u_{ik} = u_i$. In the following lemma, we show that the optimal solution of the new instance is at least $\mathcal{OPT}$ and can be transformed into a feasible solution of the original problem without losing too much profit.

Lemma 7. *The optimal allocation of the new instance is at least $\mathcal{OPT}$. Any allocation with cost $\mathcal{SOL}$ of the new instance can be transformed into a feasible solution of the original instance with cost at least $(\frac{1}{2} - \epsilon)\mathcal{SOL}$ in polynomial time.*

Proof. The first part follows easily from the fact that any feasible allocation of the original instance is also feasible in the new instance. Now, we prove the second part. An allocation $\mathbf{x}$ for the new instance may be infeasible for the original instance if we use items from multiple classes out of $\mathbb{C}_{i1}, \ldots, \mathbb{C}_{ik}$. Assume $h_i = \max\{j \mid \mathbf{x}_{ij} > 0\}$ for all i. We can obtain a feasible allocation for the original problem by only using items from $\mathbb{C}_{ih_i}$ for all i. The loss of profit can be bounded by

$$\sum_i \sum_{j=1}^{h_i-1} u_{ij} \leq \frac{1}{1-\epsilon} u_{i(h_i-1)} \leq \frac{1}{1-\epsilon} l_{ih_i} \leq \frac{1}{1-\epsilon} \mathbf{x}_{ih_i}.$$

The profit we can keep is at least $\sum_i \mathbf{x}_{ih_i}$. This proves the lemma. □

For each class $\mathbb{C}_i$, let t_i be the largest integer of the form $\lfloor (1+\epsilon)^k \rfloor$ that is at most u_i. Let the new upper bound be $\max(l_i, t_i)$. We can easily see that after the modification of u_i, the optimal value of the new instance is at least $(1-\epsilon)\mathcal{OPT}$. Moreover, we have the following property: For each positive integer T, let $\mathsf{M}(T) = \{\mathbb{C}_i \mid l_i \leq T \leq u_i\}$. Because for all $\mathbb{C}_i \in \mathsf{M}_T$, u_i is either equal to l_i (also equal to T) or in the form of $\lfloor (1+\epsilon)^k \rfloor$, and $u_i/l_i \leq 1/\epsilon$, we have the following property:

P1. All classes in M_T has at most $O(\log_{1+\epsilon} \frac{1}{\epsilon}) = O(1)$ different upper bounds any any fixed constant $\epsilon > 0$.

Corollary 1. *Assuming $|\mathbf{N}|$ is polynomially bounded, a polynomial time exact algorithm for the simplified instance (satisfying P1) implies a $\frac{1}{2} - \epsilon$ for the original instance for any constant $\epsilon > 0$.*

C.2 An Dynamic Program For the Simplified Instance

Now, we present a dynamic program for the simplified instance. Our dynamic program runs in time $\text{poly}(M, K, |\mathbf{N}|)$.

We first need to define a few notations. We use $\mathbf{N}(i, y)$ to denote the vector

$$\{N_1, N_2, \ldots, N_{i-1}, N_i + y\}$$

for every k and x. Each $\mathbf{N}(k, x)$ will play the role of the target vector for some subinstance in the dynamic program. Note that the original target vector $\mathbf{N}$ can be written as $\mathbf{N}(n,0)$. For each positive integer T, we let $\mathsf{M}_T = \{\mathbb{C}_i \mid l_i \le T \le u_i\}$ and $\mathsf{M}_T^t = \{\mathbb{C}_i \mid l_i \le T \le u_i = t\}$. Note that M_T^t are not empty for $O(\log M)$ different t due to P1. Let $\mathsf{A}_T^t = \{\mathbb{C}_i \mid l_i = T, u_i = t\}$ and $\mathsf{B}_T^t = \{\mathbb{C}_i \mid l_i < T, u_i = t\}$. Note that $\mathsf{M}_T = \cup_t \mathsf{M}_T^t$ and $\mathsf{M}_T^t = \mathsf{A}_T^t \cup \mathsf{B}_T^t$. Due to P1, we can see that for a fixed T, there are at most $O(1)$ different t values such that A_T^t and B_T^t are nonempty.

We define a subinstance $I(T, \{A_t\}_t, \{B_t\}_t, k, x)$ with the following interpretation of the parameters:

1. T: We require $\mathbf{x}_i \le T$ for all i in the subinstance.
2. $\{A_t\}_t, \{B_t\}_t$: Both $\{A_t\}$ and $\{B_t\}$ are collections of subsets of classes. Each $A_t \subset \mathsf{A}_T^t$ ($B_t \subset \mathsf{B}_T^t$ resp.) consists of the least profitable $|A_t|$ classes in A_T^t ($|B_t|$ classes in B_T^t resp.). We require that among all classes in A_T^t (B_T^t resp.), only those classes in A_t (B_t resp.) may participate in the solution. If $\mathbb{C}_i$ participates in the solution, we must have $l_i \le \mathbf{x}_i \le u_i$. Basically, $\{A_t\}, \{B_t\}$ capture the subset of classes in M_T that may participates in the solution of the subinstance. Since there are at most $O(1)$ different t such that A_T^t and B_T^t are nonempty, we have at most $n^{O(1)}$ such different $\{A_t\}$s and $\{B_t\}$s (for a fixed T).
3. Each class $\mathbb{C}_i$ with $u_i < T$ may participate in the solution.
4. k, y: $\mathbf{N}(k, y)$ is the target vector for the subinstance.

We use $\mathcal{OPT}(T, \{A_t\}, \{B_t\}_t, k, x)$ to denote the optimal solution for the subinstance $I(T, \{A_t\}, \{B_t\}_t, k, x)$.

Now, we present the recursions for the dynamic program. In the recursion, suppose we want to compute the value $D(T, \{A_t\}, \{B_t\}, i, x)$. Let $\mathbb{C}_{a(t)}$ be the A_tth least profitable class in A_T^t and $\mathbb{C}_{b(t)}$ be the B_tth least profitable class in B_T^t. We abuse the notation $\{A_{t'} - 1\}_t$ to denote the same set as $\{A_t\}_t$ except that the subset $A_{t'}$ is replaced with $A_{t'} \setminus \mathbb{C}_{a(t)}$ (i.e., the most profitable class in $A_{t'}$ is removed). The value of $D(T, \{A_t\}, \{B_t\}, \beta, i, y)$ can be computed as follows:

$$\max_{t'} \begin{cases} D(T, \{A_{t'} - 1\}_t, \{B_t\}_t, i-1, y + N_i - T) + p_{a(t)}T, & \text{if } A_t > 1 \wedge y + N_i - T \ge 0; \quad \text{(A)} \\ D(T, \{A_t\}_t, \{B_{t'} - 1\}_t, i-1, y + N_i - T) + p_{b(t)}T, & \text{if } B_t > 1 \wedge y + N_i - T \ge 0; \quad \text{(B)} \\ D(T-1, \{A'_t\}_t, \{B'_t\}_t, i, y), & \text{see explanation below;} \quad \text{(C)} \end{cases}$$

(A) captures the case that $\mathbb{C}_{a(t)}$ participates in the optimal solution of the subinstance and $\mathbf{x}_{a(t)} = T$. Similarly, (B) captures the decision $\mathbf{x}_{b(t)} = T$. In case (C), $\{A'_t\}_t, \{B'_t\}_t$ are obtained from $\{A_t\}_t, \{B_t\}_t$ as follows:

1. For any $t \ge T$ and $|B_t| > 0$, let $A'_t \subset B_t$ be the set of classes with lower bound $T-1$ and B'_t be $B_t \setminus A'_t$.
2. We need to include all classes with upper bound $T-1$. That is to let $A'_{T-1} = \mathcal{A}_{T-1}^{T-1}$ and $B'_{T-1} = \mathcal{B}_{T-1}^{T-1}$.

Note that this construction simply says $\{A'_t\}$ and $\{B'_t\}$ should be consistent with $\{A_t\}$ and $\{B_t\}$. (C) captures the case that $\mathbf{x}_i < T$ for all $i \in \cup_t \mathsf{A}_T^t \cup \mathsf{B}_T^t$. This finishes the description of the dynamic program. We can see the dynamic program runs in time poly$(M, K, |\mathbf{N}|)$ since there are at most $O(|\mathbf{N}|^2 M^{O(1)})$ different subinstances and computing the value of each subinstance takes constant time.

Now, we show why this dynamic program computes the optimal value for all subinstances defined above. In fact, by a careful examination of the dynamic program, we can see that it suffices to show the following two facts in subinstance $I(T, \{A_t\}, \{B_t\}, k, y)$. Recall $\mathbb{C}_{a(t)}$ is the A_tth cheapest class in A_T^t and $\mathbb{C}_{b(t)}$ is the B_tth cheapest class in B_T^t. Fix some $t \geq T$, we have that

1. Either $\mathbf{x}_i = 0$ for all $i \in \mathsf{A}_T^t$ or $\mathbf{x}_{a(t)} = T$.
2. Either $\mathbf{x}_i < T$ for all $i \in \mathsf{B}_T^t$ or $\mathbf{x}_{b(t)} = T$.

For each $i \in \mathsf{A}_T^t$, we can have either $\mathbf{x}_i = 0$ or $\mathbf{x}_i = T$ (since both the lower and upper bounds are T). Hence, if some $\mathbf{x}_i$ is set to be T, it is better to be the most profitable one in A_t, i.e., $\mathbb{C}_{a(t)}$. This proves the first fact. To see the second fact, suppose that $\mathbf{x}_{b(t)} < T$ but $\mathbf{x}_j = T$ for some cheaper $\mathbb{C}_j \in \mathsf{B}_T^t(B_t)$ (i.e., $p_j < p_{b(t)}$) in some optimal solution of the subinstance. By increasing $\mathbf{x}_{b(t)}$ by 1 and decreasing $\mathbf{x}_j$ by 1, we obtain another feasible allocation with a strictly higher profit, contradicting the optimality of the current solution. Having proved the correctness of the dynamic program, we summarize our result in this subsection in the following lemma.

Lemma 8. *There is a polynomial time algorithm for the problem if the given instance satisfies P1.*

Combining Corollary 1 and Lemma 8, we obtain the main result in this section. Our algorithm runs in pseudo-polynomial time. To make the algorithm runs in truly polynomial time, we can use the technique developed in Section A using the modified profits as one dimension of the dynamic program instead of using the total size. The profit loss incurred in this step can be bounded by $\epsilon \cdot \mathcal{OPT}$ in the same way. The details are quite similar to those in Section A and we omit them here.

Theorem 3. *For any constant $\epsilon > 0$, there is a factor $\frac{1}{2} - \epsilon$ approximation algorithm for CAP even when the target vector $\mathbf{N}$ is not monotone.*