Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

Lagrangian Decomposition Algorithm for Allocating Marketing Channels

Daisuke Hatano, Takuro Fukunaga, Takanori Maehara, Ken-ichi Kawarabayashi
National Institute of Informatics
JST, ERATO, Kawarabayashi Large Graph Project
{hatano,takuro,maehara,k_keniti}@nii.ac.jp

Abstract

In this paper, we formulate a new problem related to the well-known influence maximization in the context of computational advertising. Our new problem considers allocating marketing channels (e.g., TV, newspaper, and websites) to advertisers from the view point of a match maker, which was not taken into account in previous studies on the influence maximization. The objective of the problem is to find an allocation such that each advertiser can influence some given number of customers while the slots of marketing channels are limited.

We propose an algorithm based on the Lagrangian decomposition. We empirically show that our algorithm computes better quality solutions than existing algorithms, scales up to graphs of 10M vertices, and performs well particularly in a parallel environment.

1 Introduction

A major decision in a marketing plan deals with the allocation of a given budget among marketing channels, such as TV, newspaper, and websites, in order to maximize the impact on a set of potential customers. This problem can be formulated as follows. Suppose that we have estimates for the extent to which marketing channels can influence customer decisions and convert potential customers into loyal buyers, and that we would like to market a new product that will possibly be adopted by a large fraction of the customers. How should we choose a few influential marketing channels that can provoke a cascade of influence?

This problem is closely related to the well-known "*influence maximization*" problem that finds a small set of the most influential individuals in a social network so that their aggregated influence in the network is maximized. The seminal work by (Kempe, Kleinberg, and Tardos 2003) provides the first systematic study of influence maximization as a combinatorial optimization problem. The influence maximization problem further motivated the research community to conduct extensive studies on various aspects of these problems (e.g., (Chen, Wang, and Wang 2010; Chen, Wang, and Yang 2009; Borgs et al. 2014; Romero, Meeder, and Kleinberg 2011; Du et al. 2013)).

The above-mentioned problem for marketing channels was first considered by (Alon, Gamzu, and Tennenholtz 2012) who proposed the influence maximization problem with budget constraints (which is called a (*source-node*) *bipartite influence model*). Specifically, we may model the problem as a bipartite graph in which one side is the set of possible marketing channels and the other side is the set of customers. An edge between a channel i and a customer j indicates that i may influence j with some probability that depends on the budget allocated to i. There are a few more constraints in this model, and we shall provide more details in the next subsection. This model was extended by (Soma et al. 2014).

In this paper, we consider a different problem. In the context of computational advertising, three participants come into play; namely advertisers, customers, and publishers (= marketing channels, who make money by showing advertisements). The purpose of advertisers is to maximize the influence on customer decisions and then convert potential customers into loyal buyers, subject to budget constraints. However in practice, the slots for publishers are limited and moreover publishers need to increase impressions/clicks from users (so they want to display many different advertisements). Therefore a "match maker," who allocates the slots to advertisers appropriately, is desperately needed for these three participants. This fact motivates numerous previous studies on advertisement allocations (e.g., (Feldman et al. 2010; Goel et al. 2010)). However, in the (bipartite) influence maximization problem, this aspect was not previously taken into account; therefore, our purpose here is to model the above-mentioned problem in terms of a "match maker." Specifically, we consider the following conditions;

1. In order for all advertisers to be satisfied, we seek to guarantee that each advertiser will convert some given number of customers into loyal buyers in expectation.
2. In order to allocate some very "influential" marketing channel to advertisers as fairly as possible, for each marketing channel, we impose some "upper bound" of budget for each advertiser.
3. We limit the number of available slots for each marketing channel.

Our purpose is to find a solution satisfying the above three conditions. To the best of our knowledge, this is the first time the influence maximization problem has been considered in terms of a "match maker." Let us now formulate our problem

more precisely.

Bipartite influence model

Let us first define our bipartite influence model. We denote the sets of non-negative integers and non-negative reals by $\mathbb{Z}_+$ and $\mathbb{R}_+$, respectively. For $i \in \mathbb{Z}_+$, let $[i]$ denote $\{1, \ldots, i\}$. The bipartite influence model is defined as follows. Let $(S, T; E)$ be a bipartite graph, where (S, T) is a bipartition of the vertex set and $E \subseteq S \times T$ is the edge set of the graph. Vertices in S and T are called *source vertices* and *target vertices*, respectively. Source vertices correspond to marketing channels, and target vertices represent customers. Each source vertex s is associated with $c(s) \in \mathbb{Z}_+$, which represents the number of available slots of the marketing channel corresponding to s. Each $st \in E$ is associated with a probability $p(st) \in [0, 1]$, which means that putting an advertisement to a slot of s activates customer t with probability $p(st)$.

We now explain the constraints that are necessary for our model. Assuming that there are k players, we consider allocating all available slots to them. Suppose that player i is allocated $x_i(s)$ slots of each $s \in S$. An upper-bound vector $u_i \in \mathbb{Z}_+^S$ is given for each player i, and we have a constraint that $x_i \leq u_i$ (i.e., $x(s) \leq u_i(s)$ for all s), which implies that we cannot allocate more than $u_i(s)$ slots of s to player i. By putting the advertisements to the allocated slots, each player i seeks to activate target vertices. We assume that the activation events are independent. Hence the expected number of target vertices activated by player i is

$$f(x_i) = \sum_{t \in T} \left\{ 1 - \prod_{st \in E} (1 - p(st))^{x_i(s)} \right\}. \tag{1}$$

We let $\theta_i \in \mathbb{R}_+$ denote the expected number of target vertices which player i wishes to activate. Having defined this model, our goal is to find $x_1, \ldots, x_k \in \mathbb{Z}_+^S$ such that $\sum_{i=1}^k x_i(s) \leq c(s)$ for each $s \in S$, and $f(x_i) \geq \theta_i$ and $x_i \leq u_i$ for each $i \in [k]$. Here, these three constraints correspond to the three conditions mentioned at the end of the previous subsection.

However, it possibly happens that no solution satisfies all of the conditions. Indeed, when the capacities are small and θ_i is large, we cannot satisfy $f(x_i) \geq \theta_i$ even if we allocate all slots to player i. Hence, we instead consider an alternative optimization problem that is obtained by relaxing the second constraint, where the objective is to minimize the total violation on the relaxed constraints (i.e., $\sum_{i=1}^k \max\{\theta_i - f(x_i), 0\}$). Note that this objective is equivalent to maximizing $\sum_{i=1}^k \min\{f(x_i), \theta_i\}$. In summary, the optimization problem is formulated as follows:

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^k \min\{f(x_i), \theta_i\} \\ \text{subject to} & \sum_{i=1}^k x_i(s) \leq c(s) \text{ for each } s \in S, \\ & x_i \leq u_i \text{ for each } i \in [k], \\ & x_1, \ldots, x_k \in \mathbb{Z}_+^S. \end{array} \tag{2}$$

Submodular influence model

Let $g \colon \mathbb{Z}_+^S \to \mathbb{R}_+$ be a function defined on an integer lattice wherein each dimension is indexed by a source vertex. For $x, y \in \mathbb{Z}_+^S$, we let $x \wedge y$ and $x \vee y$ denote the coordinate-wise minima and maxima, respectively. A function g is denoted as *monotone* if $g(x) \leq g(y)$ for any $x, y \in \mathbb{Z}_+^S$ with $x \leq y$, and g is called *submodular (over an integer lattice)* if g satisfies $g(x) + g(y) \geq g(x \wedge y) + g(x \vee y)$ for any $x, y \in \mathbb{Z}_+^S$. It is known that the function f defined in (1) is monotone submodular (see (Soma et al. 2014)). Moreover, if g is a monotone submodular function and $\theta \in \mathbb{R}_+$ is an arbitrary real number, the function g' defined by $g'(x) = \min\{g(x), \theta\}$ for $x \in \mathbb{Z}_+^S$ is also monotone submodular. Hence (2) can be extended to the following problem defined from given monotone submodular functions $g_1, \ldots, g_k$:

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^k g_i(x_i) \\ \text{subject to} & \sum_{i=1}^k x_i(s) \leq c(s) \text{ for each } s \in S, \\ & x_i \leq u_i \text{ for each } i \in [k], \\ & x_1, \ldots, x_k \in \mathbb{Z}_+^S. \end{array} \tag{3}$$

Besides the functions defined from f and θ_i, the class of monotone submodular functions includes various important functions, and hence (3) has many other applications. We refer to (Soma et al. 2014) for examples of such applications. If $c \equiv 1$ (i.e., $c(s) = 1$ for all $s \in S$) and $u_i \equiv 1$ for each $i \in [k]$, (3) is known as the *submodular welfare maximization* (Lehmann, Lehmann, and Nisan 2006), which was introduced with motivation in combinatorial auctions. Thus (3) is an extension of the submodular welfare maximization to an *integer lattice setting*.

For $\alpha \in [0, 1]$, we say that an algorithm is an α-*approximation* or achieves an *approximation ratio* α if it always computes a feasible solution whose objective value is not smaller than α times the optimal value. (Khot et al. 2008) showed that the submodular welfare maximization admits no polynomial-time algorithm of an approximation ratio better than $1 - 1/e$ unless P = NP. Since (3) extends the submodular welfare maximization, this hardness result can be also applied to (3).

Our contribution

We first reveal the basic properties of the newly formulated problems (2) and (3). In particular, we discuss the approximability of these problems. Regarding (2), we first observe that, since (2) is a special case of (3), the $(1 - 1/e)$-approximation hardness may not be applied to (2). Indeed, (2) has a useful property that is not possessed by (3) in general. Let e_s denote the S-dimensional vector such that $e_s(s') = 1$ if $s' = s$, and $e_s(s') = 0$ otherwise. We say that a function g satisfies the *diminishing marginal return property* if $g(x + e_s) - g(x) \geq g(x + 2e_s) - g(x + e_s)$ for any $x \in \mathbb{Z}_+^S$ and $s \in S$. All monotone submodular functions over integer lattices do not necessarily have this property, but it is known and is easy to check that the function f defined by (1) does.

Having mentioned this fact, our main theoretical contribution is to show that, if $g_1, \ldots, g_k$ have the diminishing marginal return property, (3) can be regarded as an instance of maximizing a submodular *set* function defined on a set of size $\sum_{i=1}^k \sum_{s \in S} u_i(s)$ subject to partition constraints (refer to Section 2 for its definition). We also observe that it is NP-hard to achieve better than a $15/16$-approximation for (2),

and that a natural continuous relaxation of (2) is a convex programming problem

For the submodular function maximization subject to partition constraints, (Călinescu et al. 2011) proposed a $(1 - 1/e)$-approximation algorithm. We can actually extend their algorithm to give $(1 - 1/e)$-approximation for (2) and (3) with submodular functions satisfying the diminishing marginal return property. However, this is not practical because it requires substantial computational time. There are two reasons for this computational expense: First, the size of the instance of the submodular function maximization depends on $\sum_{i=1}^{k} \sum_{s \in S} u_i(s)$, and therefore it is large when working with large capacities. Second, the $(1 - 1/e)$-approximation algorithm solves a non-convex programming problem using a continuous greedy algorithm, that works very slowly.

To overcome these technical difficulties, we propose a new algorithm. Our algorithm is motivated by the fact that greedy algorithms achieve good approximation ratios for various submodular function maximization problems (e.g., (Nemhauser, Wolsey, and Fisher 1978; Sviridenko 2004)). This implies that, if the submodular objective function can be separated from the capacity constraints, we can expect that a greedy algorithm gives good solutions. Now the Lagrangian decomposition comes into play.

The *Lagrangian decomposition* is a technique, which has been widely used in various optimization problems (e.g., (Hirayama 2006; Komodakis, Paragios, and Tziritas 2007; Hatano and Hirayama 2011; Rush and Collins 2014)). Its key idea is to decompose the problem into several subproblems by introducing auxiliary variables and a Lagrangian relaxation of the problem. In our case, we replace variables in the capacity constraints with auxiliary variables $y_1, \ldots, y_k$, imposing the equality constraint $x_i = y_i$ for each $i \in [k]$. The problem remains equivalent even after this modification. We then relax the equality constraints to obtain a Lagrangian relaxation problem. Since the objective function and the capacity constraints share no common variables in the relaxation problem, it is possible to decompose the problem into subproblems, for which greedy algorithms perform well.

Our algorithm is equipped with the following useful features:

- It proceeds in iterations. The first iteration computes a feasible solution, and subsequent iterations improve the quality of solutions while preserving their feasibility. Therefore, except for the first iteration, our algorithm always keeps a feasible solution.
- Each iteration solves k instances of the submodular function maximization problem using a greedy algorithm, and integrates the obtained solutions. Therefore, each iteration does not require heavy computation and is easy to implement in parallel.

We demonstrate that our algorithm is practical through computational experiments in Section 4. We empirically prove that our algorithm scales up to graphs of 10M vertices, whereas the $(1 - 1/e)$-approximation algorithm does not scale to even graphs of 1K vertices.

2 Basic properties of the problems

One purpose of this paper is to reveal the basic properties of the problems (2) and (3) that we introduced in this paper. Here are our contributions on this issue:

- If $g_1, \ldots, g_k$ have the diminishing marginal return property, (3) can be regarded as an instance of maximizing a submodular function defined on a set of size $\sum_{i=1}^{k} \sum_{s \in S} u_i(s)$ subject to partition constraints. Under this circumstance, if we tolerate the computational time depending on $\sum_{i=1}^{k} \sum_{s \in S} u_i(s)$, by extending the existing $1/2$- and $(1 - 1/e)$-approximation algorithms (Fisher, Nemhauser, and Wolsey 1978; Călinescu et al. 2011), there are approximation algorithms that can achieve the same approximation ratios for (3). As mentioned before, this is best possible because, unless P = NP, the submodular welfare maximization problem admits no polynomial-time algorithm of an approximation ratio better than $(1 - 1/e)$ (Khot et al. 2008), and therefore (3) also admits no such approximation algorithm even if $c \equiv 1$.
- The function f defined in (1) is concave if its domain is extended to real vectors. Therefore, a continuous relaxation of (2) is a convex programming problem, and we can solve the relaxed problem using generic convex programming solvers.
- (2) includes several NP-hard problems such as the partition problem, the set k-cover problem, and the hypergraph max cut problem. (Abrams, Goel, and Plotkin 2004) showed that it is NP-hard to achieve approximation ratio better than $15/16$ for the set k-cover problem. Hence (2) admits no polynomial-time $15/16$-approximation algorithm unless P=NP. In addition to this, we present several hardness results for (2).

In this section, we present only the first result. We omit the other results in this paper due to the space limitation.

Given a monotone submodular set-function $h\colon 2^S \to \mathbb{Z}_+$ on a finite set S (i.e., $h(X)+h(Y) \geq h(X \cap Y)+h(X \cup Y)$ for $X, Y \in 2^S$, and $h(X) \geq h(Y)$ for $X, Y \in 2^S$ with $X \supseteq Y$), the problem of finding a subset U of S that maximizes $h(U)$ is called the *monotone submodular set-function maximization*. Let $\{S_1, \ldots, S_m\}$ be a partition of S (i.e., $S_i \cap S_j = \emptyset$ for $i \neq j$ and $\bigcup_{i=1}^{m} S_i = S$), and $w : \{1, \ldots, m\} \to \mathbb{Z}_+$. The constraints $|U \cap S_i| \leq w(i)$, $i = 1, \ldots, m$ are called the *partition constraints*. For the monotone submodular set-function maximization subject to partition constraints, (Fisher, Nemhauser, and Wolsey 1978) showed that a greedy algorithm achieves a $1/2$-approximation ratio, and (Călinescu et al. 2011) proposed a $(1 - 1/e)$-approximation algorithm. We explain that (3) can be reduced to the submodular set-function maximization subject to partition constraints when $g_1, \ldots, g_k$ have the diminishing marginal return property. In the reduction, the submodular set function is defined on a set of size $\sum_{i=1}^{k} \sum_{s \in S} u_i(s)$. Therefore, it does not give a polynomial-time approximation algorithm, but a pseudo-polynomial time one.

Theorem 1. *Suppose that $g_1, \ldots, g_k$ satisfy the diminishing marginal return property. If the submodular set-function*

maximization subject to partition constraints admits an α-approximation polynomial time algorithm, then (3) *admits an α-approximation pseudo-polynomial time algorithm.*

Proof. Define $\tilde{S}$ as $\{(s,i,l)\colon s \in S, i \in [k], l \in [u_i(s)]\}$. For each $U \in 2^{\tilde{S}}$ and $i \in [k]$, we let $\chi_{U,i} \in \mathbb{Z}_+^S$ denote the vector such that $\chi_{U,i}(s) = |U \cap \{(s,i,l)\colon l \in [u_i(s)]\}|$ for each $s \in S$. Define $\tilde{g}_i\colon 2^{\tilde{S}} \to \mathbb{R}_+$ as a function such that $\tilde{g}_i(U) = g_i(\chi_{U,i})$ for $U \in 2^{\tilde{S}}$. Then, (3) is equivalent to

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^k \tilde{g}_i(U) \\ \text{subject to} & \sum_{i=1}^k \chi_{U,i} \le c, \\ & U \in 2^{\tilde{S}}. \end{array} \tag{4}$$

$\tilde{g} = \sum_{i=1}^k \tilde{g}_i$ is a monotone submodular set-function. Each element in $\tilde{S}$ does not appear in more than one capacity constraint in (4). In summary, (4) is a submodular set-function maximization subject to partition constraints. □

We briefly illustrate how the above approximation algorithms for the submodular set-function maximization subject to the partition constraints compute solutions to (4). Given a current solution set U, the greedy algorithm selects $(s,i,l) \in \tilde{S}$ such that $U \cup \{(s,i,l)\}$ is a feasible solution of (4) and has the maximum gain $\tilde{g}(U \cup \{(s,i,l)\}) - \tilde{g}(U)$, and adds it to U. The algorithm iterates this operation until no such (s,i,l) exists, and then outputs the final solution set U. (Fisher, Nemhauser, and Wolsey 1978) showed that this greedy algorithm achieves a $1/2$-approximation ratio.

The algorithm of (Călinescu et al. 2011) needs more complicated computation. The multilinear extension of $\tilde{g}$ is the function $\tilde{g}'\colon [0,1]^{\tilde{S}} \to \mathbb{R}_+$ defined by $\tilde{g}'(x) = E[\tilde{g}(U)]$ for each $x \in [0,1]^{\tilde{S}}$, where U is the set that contains $(s,i,l) \in \tilde{S}$ with probability $x(s,i,l)$. The algorithm first relaxes (4) to a non-linear programming problem by replacing $\tilde{g}$ with its multilinear extension $\tilde{g}'$. (Călinescu et al. 2011) proved that a continuous greedy algorithm computes a $(1-1/e)$-approximate solution for the non-linear programming problem; starting from the all-zero vector, the continuous greedy algorithm repeatedly updates the current solution $x \in [0,1]^{\tilde{S}}$ to $x + \delta v$, where $\delta \in [0,1]$ is a sufficiently small step size, and $v \in [0,1]^{\tilde{S}}$ is a vector that satisfies the given partition constraints and maximizes $v^\top \bigtriangledown \tilde{g}'(x)$. After computing the solution for the non-linear programming problem, the algorithm transforms it into a feasible solution to (4) by using the *pipage rounding*, which was invented by (Ageev and Sviridenko 2004). (Călinescu et al. 2011) showed that this rounding step does not decrease the objective value of the solution.

An advantage of this algorithm of (Călinescu et al. 2011) is to have the tight theoretical approximation guarantee. On the other hand, a disadvantage is its substantial computational time. There are two sources of this computational burden. One source is the reduction to the submodular set-function maximization. Since the size of $\tilde{S}$ is $\sum_{s \in S} \sum_{i=1}^k u_i(s)$, even if an algorithm runs in polynomial-time for the submodular set-function maximization, it requires a pseudo-polynomial time for (3). The other source is the continuous greedy algorithm. To obtain a $(1-1/e)$-approximate solution for the non-linear programming problem, the continuous greedy algorithm requires many iterations. In addition, v is decided from $\bigtriangledown \tilde{g}'(x)$, but computing $\bigtriangledown \tilde{g}'(x)$ requires numerous samplings because $\bigtriangledown \tilde{g}'(x)$ has no compact description evaluated in polynomial time.

3 Lagrangian decomposition algorithm

In this section, we propose an algorithm for (3) based on the Lagrangian decomposition approach introduced by (Guignard and Kim 1987). We first transform (3) into another equivalent problem by introducing auxiliary variables $y_1, \ldots, y_k$:

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^k g_i(x_i) \\ \text{subject to} & \sum_{i=1}^k y_i(s) \le c(s) \text{ for each } s \in S, \\ & x_i = y_i \le u_i \text{ for each } i \in [k], \\ & x_1, \ldots, x_k, y_1, \ldots, y_k \in \mathbb{Z}_+^S. \end{array} \tag{5}$$

This transformation aims to decompose the problem structure. Indeed, the objective function and the capacity constraints in (5) share no common variables, and they are combined through the equality constraints $x_i = y_i$, $i \in [k]$.

We then relax the equality constraints by introducing Lagrangian multipliers $\lambda_1, \ldots, \lambda_k \in \mathbb{R}^S$. The problem is reduced to the following:

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^k g_i(x_i) - \sum_{i=1}^k \lambda_i^\top (x_i - y_i) \\ \text{subject to} & \sum_{i=1}^k y_i(s) \le c(s) \text{ for each } s \in S, \\ & x_i \le u_i \text{ for each } i \in [k], \\ & y_i \le u_i \text{ for each } i \in [k], \\ & x_1, \ldots, x_k, y_1, \ldots, y_k \in \mathbb{Z}_+^S. \end{array} \tag{6}$$

The constraints on $y_1, \ldots, y_k$ in (6) are same as those on $x_1, \ldots, x_k$ in (3). Hence, $y_1, \ldots, y_k$ forms a feasible solution to (3) if they are a part of a feasible solution to (6). Moreover, by the duality of the Lagrangian relaxations, the optimal value of (6) upper-bounds that of (3) for any set of Lagrangian multipliers $\lambda_1, \ldots, \lambda_k$. If a solution to (6) satisfies $x_i = y_i$ for each $i \in [k]$, then its objective value in (6) is equal to $\sum_{i=1}^k g_i(y_i)$, which is the objective value attained by $y_1, \ldots, y_k$ in (3). These relationships imply that an α-approximate solution to (6) satisfying the equality constraints also achieves α-approximation ratio for (3).

Our algorithm iteratively solves (6) by varying the values of the Lagrangian multipliers. Each iteration consists of three steps: (i) fixing $y_1, \ldots, y_k$ and $\lambda_1, \ldots, \lambda_k$, solve (6) with respect to $x_1, \ldots, x_k$; (ii) fixing $x_1, \ldots, x_k$ and $\lambda_1, \ldots, \lambda_k$, solve (6) with respect to $y_1, \ldots, y_k$; (iii) update $\lambda_1, \ldots, \lambda_k$ for the next iteration. In Steps (i) and (ii), the algorithm tries to obtain near-optimal solutions for (6) by greedy algorithms. We cannot provide a theoretical guarantee on their solution qualities, but we can expect that the greedy algorithm outputs good solutions (see later). In Step (iii), the algorithm adjusts the Lagrangian multipliers so that the solutions computed in Steps (i) and (ii) minimize the violations of the equality constraints.

We now explain Step (i) in detail. Since $y_1, \ldots, y_k$ are fixed in this step, the problem is decomposed into k disjoint

subproblems, each of which corresponds to player $i \in [k]$:

$$\text{Maximize } g_i(x_i) - \lambda_i^\top x_i \text{ subject to } x_i \leq u_i, x_i \in \mathbb{Z}_+^S. \quad (7)$$

The objective function of (7) is submodular if g_i is so. Hence (7) is a submodular function maximization problem with a simple upper-bound constraint. However, since the objective function possibly takes negative values, the problem is more difficult than maximizing non-negative submodular functions. (Kleinberg, Papadimitriou, and Raghavan 2004) and (Feige et al. 2013) analyzed greedy algorithms for this maximization problem, and presented approximation guarantees parameterized by difficulty of the problem instances. We solve (7) by the greedy-rate algorithm, which was proven in (Feige et al. 2013) to achieve the tight approximation guarantee. The greedy-rate algorithm first initializes x_i to the zero vector, and iterates the following greedy improvement; It computes $s \in S$ and $\delta \in \{1, \ldots, u_i(s) - x_i(s)\}$ that maximizes $(\Delta - \delta\lambda_i(s))/\Delta$ where Δ is the gain of $g_i(x_i)$ when $x_i(s)$ is increased by δ; If $\Delta - \delta\lambda_i(s) > 0$, it increases $x_i(s)$ to $x_i(s) + \delta$, and otherwise it terminates the iteration.

In Step (ii), the problem is given by

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^k \lambda_i^\top y_i \\ \text{subject to} & \sum_{i=1}^k y_i(s) \leq c(s) \text{ for each } s \in S, \\ & y_i \leq u_i \text{ for each } i \in [k], \\ & y_1, \ldots, y_k \in \mathbb{Z}_+^S. \end{array} \quad (8)$$

The optimal solution of (8) can be obtained as follows. For each $s \in S$, we first sort $1, \ldots, k$ into $i_1, \ldots, i_k$ so that $\lambda_{i_1}(s) \geq \cdots \geq \lambda_{i_k}(s)$. Let $k' \in [k]$ be the largest integer such that $\lambda_{i_{k'}}(s) \geq 0$. We then assign $c(s)$ units greedily to $y_{i_1}(s), \ldots, y_{i_{k'}}(s)$, i.e., we first assign $u_{i_1}(s)$ units to $y_{i_1}(s)$, and then assign $u_{i_2}(s)$ units to $y_{i_2}(s)$, and so on. We continue this process until all units are assigned, or all capacity constraints for $(i_1, s), \ldots, (i_{k'}, s)$ get tight. This gives the optimal solution for (8).

Finally, in Step (iii), we update λ_i by $\lambda_i \leftarrow \lambda_i + \eta(x_i - y_i)$ for each $i \in [k]$, where $\eta \in \mathbb{R}_+$ denotes a specified step size. In this paper, we update η in a conventional way as follows. Let UB indicate an upper bound $\sum_{i=1}^k \max_{x_i} g_i(x_i)$ of the objective value of (3); When $g_i(x_i) = \min\{f(x_i), \theta_i\}$, we set UB to $\sum_{i=1}^k \theta_i$. Let LB be the maximum objective value of the solutions found in the earlier iterations of the algorithm. Given a convergence sequence $\alpha_1, \alpha_2, \ldots$, we set $\eta = \alpha_\tau(\text{UB} - \text{LB})/(\sum_{i=1}^k \|x_i - y_i\|^2)$ in the τ-th iteration. This setting is called *subgradient optimization method*, and is known to guarantee that the Lagrange multiplier converges to the dual optimal under some conditions (Held, Wolfe, and Crowder 1974; Bertsekas 1999). In our implementation of the algorithm, we used $\alpha_\tau = 2/\sqrt{\tau}$, which yielded the best convergence in our experiments.

For each iteration, our algorithm performs greedy algorithms for solving k instances of (7), and one instance of (8). Each greedy algorithm runs in $O(|S| \sum_{i=1}^k \sum_{s \in S} u_i(s))$ time. Accordingly, the entire algorithm requires $O(k|S| \sum_{i=1}^k \sum_{s \in S} u_i(s))$ time. We emphasize that our algorithm has an advantage in scalability because the subproblems (7) can be solved in parallel; if M processors are available, the runtime is reduced to $O(\lfloor k/M \rfloor |S| \sum_{i=1}^k \sum_{s \in S} u_i(s))$.

4 Experiments

In this section, we present experimental results to evaluate our Lagrangian decomposition algorithm in terms of the solution quality, scalability, and parallel performance. To clarify the performance, we compare the Lagrangian decomposition algorithm with other algorithms. As we mentioned, the continuous greedy algorithm requires substantial computation time whereas it has the best theoretical guarantee on the solution quality. Indeed, our implementation of the continuous greedy algorithm cannot be applied to larger instances due to the time limitation. Furthermore, our additional experiments indicate that the solution quality of the greedy algorithm is not inferior to that of the continuous greedy algorithm in practice. Because of these reasons, we present only the experimental results comparing the Lagrangian decomposition algorithm with the greedy algorithm in the following subsections.

Setting of experiments

We conducted experiments on a CentOS server with Intel Xeon E5-2670@2.6GHz and 512GB of memory. The algorithms are implemented in Java and compiled with JDK 1.7.0_55. We applied our implementations to (2).

We prepared three kinds of bipartite graphs: regular graphs, power law graphs, and real dataset graphs. Since the former two kinds of graphs are constructed artificially, we can control their sizes as we wish. The third kind of graphs are constructed from open-advertising-dataset (https://code.google.com/p/open-advertising-dataset/) of query-click logs, which captures a certain situation in computational advertising. Since the dataset includes two logs, we constructed two real dataset graphs from them, one consists of 541 source vertices, 4,271 target vertices, and 5,510 edges, and the other consists of 757 source vertices, 5,062 target vertices, and 8,146 edges.

As for the capacities, $u_i(s)$ was set to 1 for each $i \in [k]$ and $s \in S$. We defined four sets of the source capacity c, which we call random, low, middle, and high settings, respectively. In the random setting, for each $s \in S$, we chose μ uniformly at random from $\{0.1, 0.2, \ldots, 0.9\}$, and defined $c(s)$ as $k \times \mu$. In the low, middle, and high settings, we chose μ from $\{0.1, 0.2, 0.3\}$, $\{0.4, 0.5, 0.6\}$, and $\{0.7, 0.8, 0.9\}$, respectively. Similarly, we constructed four sets of the targets θ_i, $i \in [k]$. In each set, we chose μ randomly and set θ_i to $f(c) \times \mu$ for each $i \in [k]$. Since we constructed four source capacity sets and four target sets, we have 16 pairs of the source capacity sets and the target sets.

Solution quality and scalability

In this subsection, we report experimental results to compare the solution quality (objective value achieved by solutions) and the scalability of algorithms.

We used three types of instances: small-, middle-, and large-scale instances. Small- and large-scale instances are constructed from regular graphs. We also had experiments

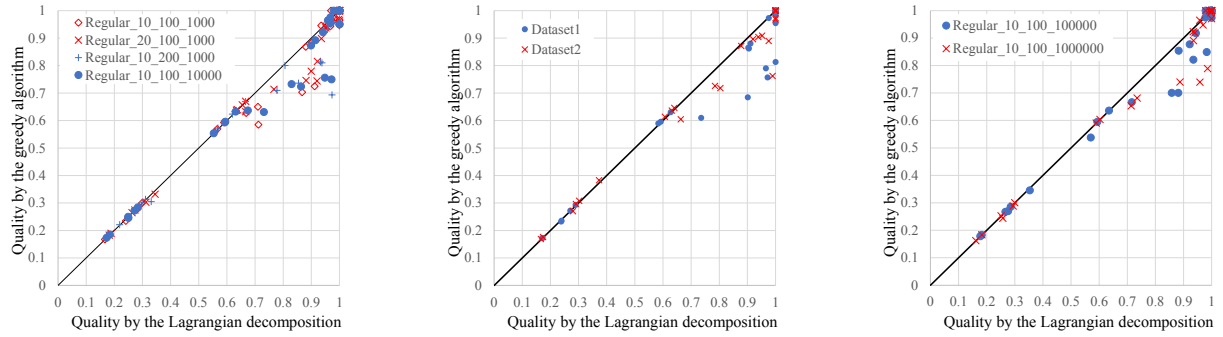

Figure 1: Solution quality over small- (left), middle- (center) and large-scale instances (right).

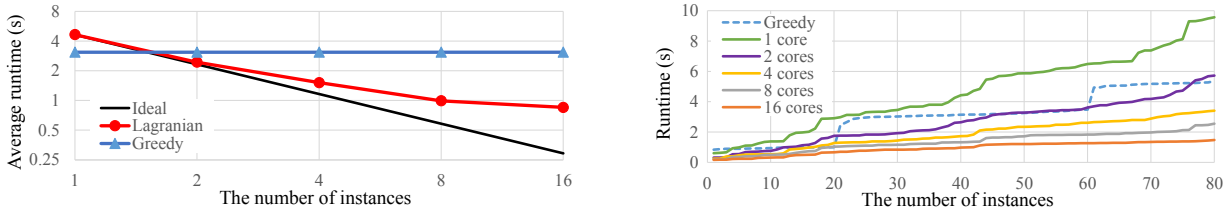

Figure 2: Parallel performance: average runtime against the number of cores (left) and runtime for each instance (right).

on small-scale instances constructed from power law graphs. Since they showed the same tendency as for regular graphs, we will report them in the full version.

We created 32 instances for each set of parameters k, $|S|$, and $|T|$; we have 16 pairs of capacity and target sets as mentioned above, and we created two regular graphs for each parameter set (this is for preparing the same number of instances as the real dataset graphs). In the small-scale instances, $|S|$ was set to 100 or 200, and $|T|$ was set to $1,000$ or $10,000$. In the large-scale instances, $|S|$ was set to 100, and $|T|$ was set to $1,000,000$. Middle-scale instances are created from the real dataset graphs; since we have two real dataset graphs and 16 pairs of source capacity and target sets, we also have 32 middle-scale instances. We set the number of iterations in the Lagrangian decomposition algorithm to 20. By preliminary experiments, we conclude that 20 iterations suffice for the Lagrangian decomposition algorithm to output good quality solutions.

The results on the solution quality are shown in Figure 1. An instance is labeled by "Regular_k_$|S|$_$|T|$" if its graph is a regular graph and it is constructed from parameters k, $|S|$, and $|T|$. The instances constructed from the dataset graphs are labeled by "Dataset1" or "Dataset2." The quality denotes the ratio of the objective value achieved by the computed solution to $\sum_{i=1}^{k} \theta_i$. Since $\sum_{i=1}^{k} \theta_i$ does not exceed the optimal objective value, the quality takes a value in $[0, 1]$, and the solution is better as its quality approaches 1. In the figure, x- and y-axis denote the solution qualities achieved by the Lagrangian decomposition and the greedy algorithms, respectively. Data plotted below the line indicates that the Lagrangian decomposition algorithm outperforms the greedy algorithm for an instance.

The results show that the Lagrangian decomposition algorithm computes solutions of equivalent or better quality compared with the greedy algorithm in most of the instances. As the quality achieved by the algorithms is higher, an instance has a solution satisfying target values for more players. We can observe that the Lagrangian decomposition algorithm particularly outputs better solutions than the greedy algorithm in many such instances. Moreover, it is noteworthy that the Lagrangian decomposition algorithm outputs solutions of quality nearly equal to one in many instances with the real dataset graphs. Since the real dataset graphs were constructed from real datasets in computational advertising, this result shows that our algorithm is practical.

We compare the runtime of the algorithms to investigate their scalability. The average runtime of the Lagrangian decomposition algorithm was 1.3 seconds, 88.4 seconds, and 4590 seconds on the small-, middle-, and large-scale instances, respectively. On the other hand, that of the greedy algorithm was 17.8 seconds, 253.4 seconds, and 11553 seconds on the small-, middle-, and large-scale instances, respectively. We can observe that the Lagrangian decomposition algorithm runs faster than the greedy algorithm even if it outputs better solutions.

In addition to these experiments, we also verified that the Lagrangian decomposition algorithm solves an instance with a regular graph, $k = 10$, $|S| = 100$, and $|T| = 10,000,000$. The runtime was 186480 seconds, while the one for the greedy algorithm was 434247 seconds, and the solution quality was 0.94 and 0.92, respectively. Since the number of customers can be huge in a context of computational advertising, it is important that the algorithm scales to graphs of this size.

Parallel performance

Finally, we evaluate the parallel performance of the Lagrangian decomposition algorithm. As noted in Section 3, the Lagrangian decomposition algorithm is easy to parallelize. We observed how the runtime of the algorithm is reduced as more cores are used. We created five graphs with $k = 10$, $|S| = 100$, and $|T| = 1000$, for each pair of the capacity and the target sets; We have 80 instances in total. The number of iterations in the algorithm was set to 20.

The results are shown in Figure 2. The left panel of Figure 2 shows the average runtime against the number of cores. The right panel shows the runtime for each instance, where instances are sorted in the increasing order of the runtime, and the height of the plot represents the runtime for the i-th instance when the x-coordinate is i. From the results illustrated in the right panel, we can observe that the Lagrangian decomposition algorithm is faster than the greedy algorithm in most of the instances even with two cores.

5 Conclusion

Extending the influence maximization problem, we formulated a new optimization problem, that allocates marketing channels to advertisers from the view point of a match maker. We revealed the complexity of this problem, and proposed a new algorithm based on the Lagrangian decomposition approach. An advantage of our algorithm is that it can produce a feasible solution quickly, it scales to large instances, and it can be easily parallelized. We empirically confirmed these advantages by comparing our algorithm with existing algorithms.

References

Abrams, Z.; Goel, A.; and Plotkin, S. A. 2004. Set k-cover algorithms for energy efficient monitoring in wireless sensor networks. In *IPSN*, 424–432.

Ageev, A. A., and Sviridenko, M. 2004. Pipage rounding: A new method of constructing algorithms with proven performance guarantee. *J. Comb. Optim.* 8(3):307–328.

Alon, N.; Gamzu, I.; and Tennenholtz, M. 2012. Optimizing budget allocation among channels and influencers. In *WWW*, 381–388.

Bertsekas, D. P. 1999. *Nonlinear programming*. Athena Scientific.

Borgs, C.; Brautbar, M.; Chayes, J.; and Lucier, B. 2014. Maximizing social influence in nearly optimal time. In *SODA*, 946–957.

Călinescu, G.; Chekuri, C.; Pál, M.; and Vondrák, J. 2011. Maximizing a monotone submodular function subject to a matroid constraint. *SIAM J. Comput.* 40(6):1740–1766.

Chen, W.; Wang, C.; and Wang, Y. 2010. Scalable influence maximization for prevalent viral marketing in large-scale social networks. In *KDD*, 1029–1038.

Chen, W.; Wang, Y.; and Yang, S. 2009. Efficient influence maximization in social networks. In *KDD*, 199–208.

Du, N.; Song, L.; Gomez-Rodriguez, M.; and Zha, H. 2013. Scalable influence estimation in continuous-time diffusion networks. In *NIPS*, 3147–3155.

Feige, U.; Immorlica, N.; Mirrokni, V. S.; and Nazerzadeh, H. 2013. PASS approximation: A framework for analyzing and designing heuristics. *Algorithmica* 66(2):450–478.

Feldman, J.; Henzinger, M.; Korula, N.; Mirrokni, V. S.; and Stein, C. 2010. Online stochastic packing applied to display ad allocation. In *ESA*, 182–194.

Fisher, M. L.; Nemhauser, G. L.; and Wolsey, L. A. 1978. An analysis of approximations for maximizing submodular set functions-II. *Math. Prog. Study* 8:73–87.

Goel, A.; Mahdian, M.; Nazerzadeh, H.; and Saberi, A. 2010. Advertisement allocation for generalized second-pricing schemes. *Oper. Res. Lett.* 38(6):571–576.

Guignard, M.; Kim, S. 1987. Lagrangean decomposition: A model yielding stronger Lagrangean bounds. *Math. Program.* 39(2):215–228.

Hatano, D., and Hirayama, K. 2011. Dynamic SAT with decision change costs: Formalization and solutions. In *IJCAI*, 560–565.

Held, M.; Wolfe, P.; and Crowder, H. P. 1974. Validation of subgradient optimization. *Math. Prog.* 6(1):62–88.

Hirayama, K. 2006. Distributed Lagrangean relaxation protocol for the generalized mutual assignment problem. In *AAMAS*, 890–892.

Kempe, D.; Kleinberg, J.; and Tardos, E. 2003. Maximizing the spread of influence through a social network. In *KDD*, 137–146.

Khot, S.; Lipton, R. J.; Markakis, E.; and Mehta, A. 2008. Inapproximability results for combinatorial auctions with submodular utility functions. *Algorithmica* 52(1):3–18.

Kleinberg, J. M.; Papadimitriou, C. H.; and Raghavan, P. 2004. Segmentation problems. *J. ACM* 51(2):263–280.

Komodakis, N.; Paragios, N.; and Tziritas, G. 2007. MRF optimization via dual decomposition: Message-passing revisited. In *ICCV*, 1–8.

Lehmann, B.; Lehmann, D.; and Nisan, N. 2006. Combinatorial auctions with decreasing marginal utilities. *Games and Econ. Behav.* 55:270–296.

Nemhauser, G.; Wolsey, L.; and Fisher, M. 1978. An analysis of the approximations for maximizing submodular set functions-I. *Math. Prog.* 14:265–294.

Romero, D. M.; Meeder, B.; and Kleinberg, J. M. 2011. Differences in the mechanics of information diffusion across topics: idioms, political hashtags, and complex contagion on twitter. In *WWW*, 695–704.

Rush, A. M., and Collins, M. 2014. A tutorial on dual decomposition and Lagrangian relaxation for inference in natural language processing. *CoRR* abs/1405.5208.

Soma, T.; Kakimura, N.; Inaba, K.; and Kawarabayashi, K. 2014. Optimal budget allocation: Theoretical guarantee and efficient algorithm. In *ICML*, 351–359.

Sviridenko, M. 2004. A note on maximizing a submodular set function subject to a knapsack constraint. *Oper. Res. Lett.* 32:41–43.