OPTIMAL CACHE PLACEMENT FOR AN ACADEMIC BACKBONE NETWORK

Than Nguyen Hau
Global Cybersoft JSC

Naomori Kakimura
Keio University

Ken-ichi Kawarabayashi
National Institute of Informatics, and
JST, ERATO, Kawarabayashi Large Graph Project

Yusuke Kobayashi
University of Tsukuba

Tatsuya Matsuoka
University of Tokyo

Yu Yokoi
National Institute of Informatics

(Received May 20, 2016; Revised March 21, 2017)

Abstract Deploying caches on a network is an effective way to reduce the amount of data transmitted in a network. Recently, in an academic backbone network such as SINET (the Science Information Network) in Japan, the amount of transmitted data has significantly increased. It is desired to design an efficient mechanism to allocate caches in an optimal way. In this paper, we begin by formulating a discrete optimization model to find a cache allocation that minimizes the total transmission cost. We then design two efficient algorithms to solve our proposed model. The first one makes use of the fact that a backbone network has small treewidth. The algorithm runs in polynomial time when the number of items is fixed and a graph has a bounded treewidth. The other one reduces the problem to the minimum-cost flow problem under the practical assumption that each item has at most one copy. This yields a polynomial-time combinatorial algorithm. Our numerical experiments on the real SINET network show that our algorithms can solve the cache placement problem efficiently in practice.

Keywords: Combinatorial optimization, facility planning, graph theory

1. Introduction

The Science Information Network (SINET) is an academic backbone network in Japan [3]. It is a nationwide network, connecting more than 800 universities and research institutes. SINET is now used by more than two million users for scientific research such as seismology, space science, high-energy physics, nuclear fusion, and computer science. To support research and education activities, SINET provides a variety of network services, including advanced multi-layer VPN services and on-demand services as well as a commercial Internet access service. The network can be depicted as in Figure 1 [1]. The implementation detail of SINET can be found in [30].

Recently, the amount of transmitted data through SINET has significantly increased. One reason is development of cloud computing [2]. Cloud computing technology allows research institutes (including universities) to place a virtual server or storage in a commercial cloud data center outside their campuses, while almost all used data are transmitted within a local area. This provides comfortable network environments for users, but has increased communication costs and security risks. Another recent trend surrounding universities is e-learning (see e.g., [20]). Most universities in Japan adopt learning management systems to provide better education for their students. Also, massive open online courses (MOOC) such as Coursera, Edx, Udacity, have started in Japan, and they provide rich contents including...
lecture videos and an online interactive forum. There, we are required to deliver lecture materials of high quality to tens of thousands of students all over the country. This has put heavy load on an academic backbone network, when the data is located at one place.

In this paper, we aim to resolve the above issues by cooperative caching. That is, we devise efficient centralized algorithms to allocate cache files in SINET so as to minimize communication costs. Note that SINET is monitored all the time to collect large amounts of traffic information of every interface by SNMP. We thus assume that demand of each node can be estimated in advance, and that it is possible to control cache placement in a centralized way.

The benefits of cooperative caching were investigated before in the setting of distributed file systems [9, 11, 14, 16, 19, 21]. Most previous studies have focused on minimizing access latency, and disregarded the bandwidth consumption involved, while assuming that bandwidth is abundant. However, SINET usually requires users to reserve link bandwidth by the day before the use so that users can occupy the bandwidth. Thus transmission resources are limited, and data congestion makes significant influence on data latency.

We tackle the cache placement problem in SINET in the following ways.

(i) We formulate the cache placement problem as an integer optimization problem to reduce transmission costs and bandwidth.

(ii) We develop an efficient dynamic programming algorithm using a graph structure of SINET.

(iii) We design a polynomial-time algorithm under a practical assumption by reducing to the minimum-cost flow problem.

We describe the above three approaches in more details.

We first formulate the cache placement problem as an integer optimization problem. The objective is to minimize the total transmission cost under the constraint that every demand of a node is fulfilled. Our formulation is closely related to the data placement problem [7] (the relationship will be described in Section 2). One of the differences is that we can cope with bandwidth while minimizing transmission costs. For that purpose, we additionally introduce an upper bound of the amount of transmission data through each link. This formulation makes it possible to find an optimal cache deployment that avoids
packet congestion. Such link capacity constraints are also important to analyze critical links in communication, which helps us improve a network design.

As in Figure 1, SINET has a special graph topology like a tree, though it is not exactly a tree as it has cycles. Our second result is to develop an algorithm for caching for a graph with small treewidth. The treewidth of a graph, introduced in [23], is a positive integer that measures the closeness of the graph to a tree. A graph has treewidth one if and only if it is a tree, and a smaller treewidth means being closer to a tree. The treewidth plays an important role in graph theory and computer science. It is used to prove famous conjectures in graph theory such as Wagner’s conjecture [25], and it is also used to design efficient algorithms for various graph problems (see, e.g., [5]). Since SINET is close to a tree, it is observed that SINET has small treewidth. Indeed, the treewidth of the central network of SINET, which consists of 11 core nodes and 19 core links, is just two. Also, in other countries, academic backbone networks usually have simple structures. For example, CESNET in Czech Republic began from a tree network, and has been augmented to increase reliability [27]. In this paper, we propose an efficient dynamic-programming algorithm to solve the cache placement problem when the treewidth is small. The algorithm runs in polynomial time when the number of items is a fixed constant. Note that the algorithm does not work if we have link capacity constraints. In fact, we prove that it is NP-complete to decide whether a given tree network has a feasible cache allocation even if we have only two items. Thus it is hopeless to design an algorithm (even, an approximation algorithm) that runs in time polynomial in the input size even if a graph is a tree.

Furthermore, we design a polynomial-time algorithm under a natural assumption. It is not so practical to allocate the same file at many places in a network. It would increase the risk of information leaks, and it also makes difficult to synchronize all the copies when updating data. Thus it is natural to impose the assumption on the total number of copied items. We here assume that each file can be put at no more than one place except its original file. We show that we can reduce the cache placement problem under this assumption to the minimum-cost flow problem. The minimum-cost flow problem is one of the most classical combinatorial optimization problems, which can be solved efficiently in a combinatorial way (see, e.g., [4, 26]). Thus the problem can be solved faster and more stably than the general case.

To evaluate performance of our cache placement algorithms, we thoroughly conducted simulation with the real SINET network. Our experimental results show that we can improve the transmission cost by about 50%, compared to the case without allocating cache files. Moreover, by adding a link capacity constraint, the bandwidth can be kept small without reducing the solution quality. In addition, we executed our polynomial-time algorithm using minimum-cost flow. Finally, we propose a heuristic via our network-flow-based algorithm, and evaluate its performance with different settings.

This paper is organized as follows. Section 1 describes previous related work. In Section 2, we formulate our cache placement problem as an integer optimization model. Section 3 discusses how to exploit a tree-like structure to solve the cache placement problem, and Section 4 describes how to reduce the problem to the minimum-cost flow problem under a practical assumption. In Section 5, we demonstrate our simulation results on SINET.

Related work

In the $p$-median problem, we are given a set of clients and a set of facilities. The aim is to open $p$ facilities so as to minimize the total cost to connect each client to some opened facility. The problem is NP-hard, while approximation algorithms have been studied extensively
when the cost is metric [6, 10]. When a graph is a tree, the $p$-median problem can be solved in polynomial time [29]. The $p$-median problem was generalized to the matroid median problem [17], which will be defined in Section 2. Krishnaswamy et al. [17] gave a 16-approximation algorithm, and Swamy [28] recently improved it to 8-approximation.

The facility location problem has been also investigated in the literature [13, 18]. In the problem, we are given an opening cost for each facility, instead of the total number $p$ of possible facilities, and the aim is to find a subset of facilities that minimizes the sum of the connection cost and the total opening cost. The data placement problem generalizes the facility location problem so that we have multiple items. Baev et al. [7] gave a 10-approximation algorithm.

In the context of the cache placement problem, also known as the replica placement problem or content-distribution networks (CDN), optimization models have extensively been studied. Motivated by video on demand (VoD) system, Borst et al. [9] proposed an integer optimization model to investigate performance of CDN. Most previous studies on CDN have been done on a tree or a hierarchical structure [9, 11, 16, 21]. This is in contrast to our formulation, which deals with a general graph. Mangili et al. [19] introduced an optimization model for CDN to minimize the total amount of transmitted data in a whole network. Kangasharju et al. [14] provided four heuristics to solve the object replication problem that minimizes the maximum transmission cost of nodes. Our formulation differs in that it minimizes the total transmission cost while keeping small bandwidth.

Recently, a new paradigm content-centric network (CCN) was proposed. This provides new protocols centered around the data itself. A comprehensive description of CCN can be found in [12]. Many simulation and experimental testbeds have been done in, e.g., [31]. Mangili et al. [19] proposed a performance model for CCN to compare with CDN.

2. Integer Optimization Model

In this section, we formulate our problem as an integer optimization problem. We first consider a simple model, and then impose an additional constraint on link capacities.

2.1. Cache placement problem

Let $G = (V, E)$ be a graph that represents a computer network, where $V$ is a node set and $E$ is a set of links. Each link $e$ in $E$ has length $c(e)$. The graph $G$ has core nodes, on which we can allocate cache files. A set of core nodes is denoted by $U \subseteq V$. We are also given a set $I$ of $k$ items. Each node $v$ in $V$ has a demand $d(v, i)$ for each item $i$. For simplicity, we assume that every $d(v, i)$ is positive in this section.

For a core node $u$, we can place at most $q(u)$ items as cache. If item $i$ is placed at $u$, then a node $v$ can receive item $i$ from $u$. The unit cost for the transmission is defined by the cost of the shortest route between $u$ and $v$ with respect to the length $c$ in $G$, which is denoted by $\text{dist}(u, v)$. We suppose that every node $v$ chooses a core node $u$ having item $i$ for each item $i$. Thus the cost to transmit the item $i$ from $u$ is $d(v, i) \text{dist}(u, v)$.

We aim to allocate cache files on core nodes so that the total transmission cost is minimized. The problem is formulated as follows. Here $x(u, v, i)$ is a 0-1 variable defined to be one if and only if we send item $i$ from core node $u$ to node $v$, and $y(u, i)$ is a 0-1 variable
defined to be one if and only if we put item \( i \) in core node \( u \).

\[
\begin{align*}
\text{minimize} & \quad \sum_{i \in I} \sum_{u, v \in V} d(v, i) \text{dist}(u, v) x(u, v, i), \\
\text{subject to} & \quad \sum_{u \in U} x(u, v, i) \geq 1, \quad \forall v \in V, \forall i \in I, \\
& \quad x(u, v, i) \leq y(u, i), \quad \forall u \in U, \forall v \in V, \forall i \in I, \\
& \quad \sum_{i \in I} y(u, i) \leq q(u), \quad \forall u \in U, \\
& \quad x(u, v, i) \in \{0, 1\}, \quad \forall u \in U, \forall v \in V, \forall i \in I, \\
& \quad y(u, i) \in \{0, 1\}, \quad \forall u \in U, \forall i \in I.
\end{align*}
\]

The problem has three constraints: the first one means that every node \( v \) chooses at least one core node \( u \) for each item \( i \), the second one says that if \( x(u, v, i) = 1 \) then \( y(u, i) \) has to be one, which guarantees that a core node \( u \) has an item \( i \) when \( u \) is chosen for item \( i \), and the third constraint ensures that a core node \( u \) has at most \( q(u) \) items. We call this problem (2.1) the \textit{cache placement problem}.

In practice, it is natural that at least one cache file, i.e., an original file, has been already placed at some core node for each item. This condition can be incorporated by forcing a variable \( y(u, i) \) to be one when \( u \) has an original file of \( i \).

The problem (2.1) is similar to the data placement problem [7]. The difference is that, in the latter problem, a core node can have an opening cost when used as cache while each node demands only one item. We see, however, that the problem (2.1) can be reduced to the data placement problem, and hence has a 10-approximation algorithm.

We can improve the approximation ratio by reducing the problem (2.1) to the \textit{matroid median problem} [17]. This is the problem to find a set of facilities such that the sum of connection costs (to the nearest opened facility) is minimized subject to the constraint that the set of opened facilities forms an independent set of a matroid. More precisely, we are given two finite sets \( U \) and \( V \) of facilities and clients, respectively, a cost \( c(u, v) \) to connect \( u \in U \) and \( v \in V \), and a matroid \( M \) on the set \( U \) (see, e.g., [26] for the definition of a matroid). Then the problem can be represented as the following optimization problem:

\[
\begin{align*}
\text{minimize} & \quad \sum_{u, v \in V} c(u, v) x(u, v), \\
\text{subject to} & \quad \sum_{u \in U} x(u, v) \geq 1, \quad \forall v \in V, \\
& \quad x(u, v) \leq y(u), \quad \forall u \in U, \forall v \in V, \\
& \quad \{u \mid y(u) = 1\} \text{ forms an independent set in } M, \\
& \quad x(u, v) \in \{0, 1\}, \quad \forall u \in U, \forall v \in V, \\
& \quad y(u) \in \{0, 1\}, \quad \forall u \in U.
\end{align*}
\]

Note that the \( p \)-median problem is a special case of the matroid median problem where the family of independent sets in \( M \) is the one of sets with size at most \( p \), i.e., a uniform matroid. Swamy [28] considered a generalized version of the matroid median problem, in which each facility can have an opening cost, and showed that the problem has an 8-approximation algorithm if the cost \( c \) is metric. He also showed that the data placement problem with metric cost can be reduced to this generalized matroid median problem. In the same way, we can reduce our problem (2.1) to the matroid median problem (2.2).

\textbf{Theorem 2.1.} There exists an 8-approximation algorithm for the cache placement problem.
Proof. We show that the cache placement problem can be reduced to the matroid median problem. We define a vertex set \( V' = \{(v, i) \mid v \in V, i \in I\} \) and \( U' = \{(u, i) \mid u \in U, i \in I\} \). For \( u' = (u, i) \in U' \) and \( v' = (v, j) \in V' \), set \( c(u', v') = d(v, i) \) \( \text{dist}(u, v) \) if \( i = j \), and \( c(u', v') = +\infty \) if \( i \neq j \). Define the matroid median problem with \( U', V' \) and \( c \), where the matroid will be given later. Then a feasible solution to the problem (2.1) corresponds to a subset of \( U' \) satisfying the first two conditions in (2.2) and that we can only choose \( q(u) \) elements from \( \{(u, i) \mid i \in I\} \) for each core node \( u \). The second constraint in fact forms a matroid (this is called a partition matroid [26]). Moreover, the objective value with respect to \( c \) is the same as that of (2.1), and it is metric. Thus the problem (2.1) can be reduced to the matroid median problem, which admits an \( 8 \)-approximation algorithm. Since each feasible solution of this matroid median instance corresponds to that of the original instance having the same objective value, this reduction preserves the approximation ratio. \( \square \)

2.2. Cache placement problem with link capacities

In addition to the above setting, consider a situation when each link has a capacity. That is, for each link \( e \), we are given a positive integer \( p(e) \), representing an upper bound of the amount of transmitted data through the link \( e \).

A naive implementation of this constraint is to introduce variables \( x(e, u, v, i) \) for a link \( e \), a core node \( u \in U \), a node \( v \in V \), and an item \( i \), that represents the amount of item \( i \) transmitted between \( u \) and \( v \) through \( e \). Then we can describe the capacity constraint using \( x(e, u, v, i) \)’s, together with the flow conservation constraints (cf. [19]). However, the size of the problem would become huge. Moreover, in the formulation, we are required to determine the optimal route for each pair of nodes and each item, but it seems impractical to control all the routings depending on cache allocation.

In this paper, we assume that we always take the shortest route when we transmit data between two nodes. For a pair of nodes such that there are multiple shortest routes between them, we specify one shortest route and always use it. For two nodes \( u \) and \( v \), we denote by \( P_{uv} \) the set of links in a shortest route between \( u \) and \( v \). For a link \( e \), let \( R_{e} \) be the set of pairs of nodes \( (u, v) \) such that \( P_{uv} \) contains the link \( e \). Then the problem can be described as follows.

\[
\begin{align*}
\text{minimize} & \quad \sum_{i \in I} \sum_{u, v \in V} d(v, i) \text{dist}(u, v) x(u, v, i), \\
\text{subject to} & \quad \sum_{u \in V} x(u, v, i) \geq 1, \quad \forall v \in V, \forall i \in I, \\
& \quad x(u, v, i) \leq y(u, i), \quad \forall u \in U, \forall v \in V, \forall i \in I, \\
& \quad y(u, i) \leq q(u), \quad \forall u \in U, \\
& \quad \sum_{(u, v) \in R_{e}} \sum_{i \in I} d(v, i) x(u, v, i) \leq p(e), \forall e \in E, \\
& \quad x(u, v, i) \in \{0, 1\}, \quad \forall u \in U, \forall v \in V, \forall i \in I, \\
& \quad y(u, i) \in \{0, 1\}, \quad \forall u \in U, \forall i \in I.
\end{align*}
\]

Comparing with the cache placement problem (2.1), we append the fourth constraint that each link \( e \) has at most \( p(e) \) amount of transmitted data. Note that we do not introduce new variables. The problem is called the cache placement problem with link capacities. Similarly to the cache placement problem (2.1), we can deal with an initial allocation of original files by forcing a variable \( y(u, i) \) to be one when \( u \) has an original file of \( i \).
3. Using Tree-like Structures

In this section, we discuss exploiting a tree-like structure of SINET to solve the cache placement problem with/without edge capacities. We first design a dynamic programming algorithm for the cache placement problem on trees. This algorithm can be extended to the problem on a graph with small treewidth. Finally, we show hardness of the cache placement problem with edge capacities.

3.1. Dynamic programming algorithm on trees

It is known that the p-median problem on trees can be solved in polynomial time by using a dynamic programming technique [29]. In this subsection, we show that a similar approach can also be applied to the cache placement problem. We here assume that we have no original data in the graph, but the same argument can be applied for the case with original data.

Suppose that we are given an instance of the cache placement problem, i.e., Problem (2.1) in Section 2, in which the given graph is a tree $T = (V, E)$. Fix one node $r_T \in V$ as a root. For $u, v \in V$, we say that $u$ is a descendant of $v$ if $v$ is on the unique path connecting $r_T$ and $u$. If $u$ and $v$ are adjacent and $u$ is a descendant of $v$, then $u$ is a child of $v$. A node having no children is called a leaf of the tree. For $v \in V$, let $T_v$ denote the subtree of $T$ which is induced by all descendants of $v$ (including $v$). When we consider the cache placement problem, by adding links of length zero if necessary, we may assume that each non-leaf node $v$ has exactly two children, say $v_{\text{left}}$ and $v_{\text{right}}$ (see e.g., [29]). By perturbing the link length if necessary, we may also assume that all distances between two nodes are distinct.

Recall that $k$ is the number of items. For $v \in V$, $v_1^{\text{in}}, \ldots, v_k^{\text{in}} \in V(T_v) \cup \{\emptyset\}$, and $v_1^{\text{out}}, \ldots, v_k^{\text{out}} \in (V - V(T_v)) \cup \{\emptyset\}$, we denote by $F(v, v_1^{\text{in}}, \ldots, v_k^{\text{in}}, v_1^{\text{out}}, \ldots, v_k^{\text{out}})$ the optimal value of the subproblem on the subtree $T_v$ under the following constraints:

- Among all copies of item $i$ in $V(T_v)$, the closest one to $v$ is in $v_i^{\text{in}}$. If $v_i^{\text{in}} = \emptyset$, then no node in $V(T_v)$ has item $i$, and we define $\text{dist}(v, v_i^{\text{in}}) = +\infty$.
- The cache placement in $V - V(T_v)$ is fixed, and among all copies of item $i$ in $V - V(T_v)$, the closest one to $v$ is in $v_i^{\text{out}}$. If $v_i^{\text{out}} = \emptyset$, then no node in $V - V(T_v)$ has item $i$, and we define $\text{dist}(v, v_i^{\text{out}}) = +\infty$.

We note that the demands in $V - V(T_v)$ are not considered in this subproblem. We also note that the value of $F$ can be $+\infty$.

In what follows, we give an algorithm to compute all the values of $F$. We note that the number of choices of $v, v_1^{\text{in}}, \ldots, v_k^{\text{in}}, v_1^{\text{out}}, \ldots, v_k^{\text{out}}$ is at most $|V|^{2k+1}$, which is a polynomial in $|V|$ when $k$ is a fixed constant. If we know all the values of $F$, then we can compute the optimal value of the cache placement problem, which is equal to the minimum of $F(r_T, v_1^{\text{in}}, \ldots, v_k^{\text{in}}, \emptyset, \ldots, \emptyset)$, where the minimum is taken over all $v_1^{\text{in}}, \ldots, v_k^{\text{in}} \in V$.

If $v$ is a leaf of the tree, we can easily compute $F(v, v_1^{\text{in}}, \ldots, v_k^{\text{in}}, v_1^{\text{out}}, \ldots, v_k^{\text{out}})$. Therefore, our remaining task is to compute $F(v, v_1^{\text{in}}, \ldots, v_k^{\text{in}}, v_1^{\text{out}}, \ldots, v_k^{\text{out}})$ efficiently when we have solutions of the subproblems in $T_{\text{left}} := T_{\text{left}}$ and $T_{\text{right}} := T_{\text{right}}$.

We partition $I = \{1, \ldots, k\}$ into four sets $I_0, I_\emptyset, I_{\text{left}}, I_{\text{right}}$ so that $v_i^{\text{in}} = v$ for $i \in I_0$, $v_i^{\text{in}} = \emptyset$ for $i \in I_\emptyset$, $v_i^{\text{in}} \in V(T_{\text{left}})$ for $i \in I_{\text{left}}$, and $v_i^{\text{in}} \in V(T_{\text{right}})$ for $i \in I_{\text{right}}$. If $|I_0| > q(v)$, then $F(v, v_1^{\text{in}}, \ldots, v_k^{\text{in}}, v_1^{\text{out}}, \ldots, v_k^{\text{out}}) = +\infty$. Otherwise, we can compute
\(F(v, v_{i1}^{\text{in}}, \ldots, v_{i1}^{\text{out}}, \ldots, v_{ik}^{\text{out}})\) by the following formula:

\[
F(v, v_{i1}^{\text{in}}, \ldots, v_{ik}^{\text{in}}, v_{i1}^{\text{out}}, \ldots, v_{ik}^{\text{out}}) = \min \left\{ F(v_{\text{left}}, u_{i1}^{\text{in}}, \ldots, u_{ik}^{\text{in}}, v_{i1}^{\text{out}}, \ldots, v_{ik}^{\text{out}}) \right. \\
+ F(v_{\text{right}}, u_{i1}^{\text{in}}, \ldots, v_{ik}^{\text{in}}, w_{i1}^{\text{out}}, \ldots, w_{ik}^{\text{out}}) \right. \\
\left. + \sum_{i=1}^{k} d(v, i) \min \{ \text{dist}(v, v_{i1}^{\text{in}}), \text{dist}(v, v_{i1}^{\text{out}}) \} \right.
\]

where the first minimum is taken over all \(u_{i1}^{\text{in}}, \ldots, u_{ik}^{\text{in}} \in V(T_{\text{left}}) \cup \{ \emptyset \}, w_{i1}^{\text{out}}, \ldots, w_{ik}^{\text{out}} \in (V - V(T_{\text{left}})) \cup \{ \emptyset \}, u_{i1}^{\text{in}}, \ldots, v_{i1}^{\text{in}} \in V(T_{\text{right}}) \cup \{ \emptyset \}, \) and \(w_{i1}^{\text{out}}, \ldots, w_{ik}^{\text{out}} \in (V - V(T_{\text{right}})) \cup \{ \emptyset \}\) such that

- \(u_{i1}^{\text{in}} = v_{i1}^{\text{in}}\) for \(i \in I_{\text{left}}\) and \(w_{i1}^{\text{in}} = v_{i1}^{\text{in}}\) for \(i \in I_{\text{right}}\),
- \(u_{i1}^{\text{out}} = w_{i1}^{\text{out}} = v\) for \(i \in I_0\),
- Both \(\text{dist}(u_{i1}^{\text{in}}, v)\) and \(\text{dist}(w_{i1}^{\text{in}}, v)\) are at least \(\text{dist}(v_{i1}^{\text{in}}, v)\) for any \(i \in I\),
- \(\text{dist}(v_{\text{left}}, v_{i1}^{\text{out}}) = \min \{ \text{dist}(v_{\text{left}}, w_{i1}^{\text{in}}), \text{dist}(v_{\text{left}}, v_{i1}^{\text{out}}) \}\) for \(i \in I - I_0\), which implies that either \(u_{i1}^{\text{out}} = u_{i1}^{\text{in}}\) or \(u_{i1}^{\text{out}} = v_{i1}^{\text{out}}\),
- \(\text{dist}(v_{\text{right}}, v_{i1}^{\text{out}}) = \min \{ \text{dist}(v_{\text{right}}, v_{i1}^{\text{in}}), \text{dist}(v_{\text{right}}, v_{i1}^{\text{out}}) \}\) for \(i \in I - I_0\), which implies that either \(w_{i1}^{\text{out}} = u_{i1}^{\text{in}}\) or \(w_{i1}^{\text{out}} = v_{i1}^{\text{out}}\).

See Figure 2 for an example. It is not difficult to check the correctness of this formula. We note that the number of choices of \(u_{i1}^{\text{in}}, u_{i1}^{\text{out}}, w_{i1}^{\text{in}}, w_{i1}^{\text{out}}\) for all \(i\) is at most \(|V|^4k\), which is still a polynomial in \(|V|\).

![Figure 2: An example of \(u_{i1}^{\text{in}}, u_{i1}^{\text{out}}, \) and \(v_{i1}^{\text{out}}\)](image)

By using this formula repeatedly, we can compute the values of \(F\) from the leaves to the root. Since each value is computed in \(|V|^{O(k)}\) time and we have \(|V|^{O(k)}\) choices of the indices of \(F\), the total running time to compute \(F\) is \(|V|^{O(k)}\). This shows the following theorem.

**Theorem 3.1.** The cache placement problem on a tree can be solved in \(|V|^{O(k)}\) time.

### 3.2. Extension to a graph with small treewidth

In this subsection, we extend Theorem 3.1 to a graph with bounded treewidth.

We first define the treewidth of a graph. Let \(G = (V, E)\) be a graph, and \(T\) be a tree. Let \(\mathcal{V} = \{V_t \subseteq V(G) \mid t \in V(T)\}\) be a family of node sets \(V_t \subseteq V(G)\) indexed by the nodes \(t\) of \(T\). The pair \((T, \mathcal{V})\) is called a tree-decomposition of \(G\) if it satisfies the following three conditions:

- \(V(G) = \bigcup_{t \in V(T)} V_t,\)
- for every link \(e \in E(G)\) there exists \(t \in V(T)\) such that both ends of \(e\) lie in \(V_t,\)
- if \(t, t', t'' \in V(T)\) and \(t'\) lies on the path of \(T\) between \(t\) and \(t''\), then \(V_t \cap V_{t''} \subseteq V_{t'}\).

The width of \((T, \mathcal{V})\) is the number \(\max\{\{|V_t| - 1| \ t \in V(T)|\}\) and the treewidth of \(G\) is the minimum width of any tree-decomposition of \(G\). The definition says that a graph with small
treewidth has similar structures to a tree. Note that a graph is a tree if and only if the treewidth is one.

Bodlaender [8] designed a linear-time algorithm for constructing a tree-decomposition of a graph of bounded treewidth, which was further improved in [22].

**Theorem 3.2.** For a fixed integer \( w \), there exists an \( O(n) \)-time algorithm that, given a graph \( G \), either finds a tree-decomposition of \( G \) of width at most \( w \) or concludes that the treewidth of \( G \) is more than \( w \).

Suppose that \( G \) is an input graph of the cache placement problem and we are given a tree-decomposition \((T, \mathcal{V})\) of \( G \) of width \( w \). We fix a root node of the tree \( T \). For each node set \( V_t \subseteq \mathcal{V} \) indexed by \( t \in V(T) \), we consider the subgraph \( G_t \) induced by all the nodes in \( G \) corresponding to descendants of \( t \) in \( T \). In a similar way to the tree case in Section 3.1, we define a value \( F \) whose indices consist of

- \( t \in V(T) \),
- \( v^{in}(v, i) \in V(G_t) \cup \{\emptyset\} \) and \( v^{out}(v, i) \in (V - V(G_t)) \cup \{\emptyset\} \) for each \( v \in V_t \) and \( i \in I \),
- \( X_i \subseteq V_t \) for \( i \in I \).

We define the value of \( F \) as the optimal value of the subproblem on the subgraph \( G_t \) under the following constraints.

- For each \( i \in I \), \( X_i \) is the set of nodes that have copies of item \( i \) in \( V_t \).
- Among all copies of item \( i \) in \( V(G_t) \), the closest one to \( v \) is in \( v^{in}(v, i) \). If \( v^{in}(v, i) = \emptyset \), then no node in \( V(G_t) \) has item \( i \), and we define \( \text{dist}(v, v^{in}(v, i)) = +\infty \).
- The cache placement in \( V - V(G_t) \) is fixed, and among all copies of item \( i \) in \( V - V(G_t) \), the closest one to \( v \) is in \( v^{out}(v, i) \). If \( v^{out}(v, i) = \emptyset \), then no node in \( V - V(G_t) \) has item \( i \), and we define \( \text{dist}(v, v^{out}(v, i)) = +\infty \).

Then a recursive formula holds similar to the one in Section 3.1. Since we have \(|V|^{O(kw)}\) choices of the indices of \( F \), we have the following theorem.

**Theorem 3.3.** The cache placement problem can be solved in \(|V|^{O(kw)}\) time if the treewidth of the graph is at most \( w \).

### 3.3. Hardness of the problem with link capacities

In this subsection, we prove that the cache placement problem with link capacities is NP-hard even if the graph is a tree and we have only two kinds of items.

**Theorem 3.4.** The cache placement problem with link capacities on trees is NP-hard even if we have only two items.

**Proof.** We consider the following Partition problem, which is one of Karp’s NP-complete problems [15]. For a finite set \( S \) and a function \( f : S \to \mathbb{R}_+ \), Partition is the problem to find a partition \((A, B)\) of \( S \) with \( f(A) = f(B) \), where we denote \( f(X) := \sum_{j \in X} f(j) \) for \( X \subseteq S \).

In what follows, we show that there exists a polynomial reduction from Partition to the cache placement problem with link capacities. Suppose a set \( S \) and a function \( f : S \to \mathbb{R}_+ \) are an instance of Partition. We construct an instance of the cache placement problem with link capacities as follows. Let \( G = (V, E) \) be the graph defined by

\[
V = \{u, u_1, u_2\} \cup \{v_j \mid j \in S\}, \\
E = \{(u, u_1), (u, u_2)\} \cup \{(u, v_j) \mid j \in S\},
\]

which is a tree (indeed a star). See Figure 3. For \( v \in \{u_1, u_2\} \), define \( p(u, v) = \frac{5f(S)}{2} \), \( d(v, 1) = d(v, 2) = f(S) \), and \( q(v) = 1 \). For \( j \in S \), define \( p(u, v_j) = f(j) \), \( d(v_j, 1) = \)
d(v_j, 2) = f(j), and q(v_j) = 1. We also define d(u, 1) = d(u, 2) = 0 and q(u) = 0. The length of each link is defined arbitrarily. Recall that p(u, v) is the link capacity of (u, v), d(v, i) is the demand value of item i at v, and q(v) is the number of items that can be placed at v. Note that we define U = V \ {u}, that is, we can place caches on nodes except for u.

![Graph](image)

Figure 3: Reduction to the cache placement problem with link capacities

To prove that these instances are equivalent, we consider a feasible solution of this cache placement problem. Since \( d(u_1, 1) = d(u_1, 2) = d(u_2, 1) = d(u_2, 2) = f(S) > p(u, v_j) \) for any \( j \in S \), we have to place both two items in \( \{u_1, u_2\} \). By symmetry, we may assume that item 1 is placed in \( u_1 \) and item 2 is placed in \( u_2 \). We consider the partition of \( S \) into two sets \( S_1 \) and \( S_2 \) such that item 1 is placed in \( v_j \) for any \( j \in S_1 \) and item 2 is placed in \( v_j \) for any \( j \in S_2 \).

For each \( j \in S \), since a client \( v_j \) receives a flow of value \( f(j) \) along \( (u, v_j) \), the other clients cannot use link \( (u, v_j) \). Therefore, clients in \( \{v_j \mid j \in S_1\} \cup \{u_1\} \) access \( u_2 \) to obtain item 2, and client \( u_2 \) accesses \( u_1 \) to obtain item 1, which shows that the flow value on link \( (u, u_2) \) is \( f(S_1) + 2f(S) \). Since the capacity constraint at \( (u, u_2) \) must be satisfied, we have \( f(S_1) + 2f(S) \leq p(u, u_2) \), that is, \( f(S_1) \leq \frac{f(S)}{2} \). Similarly, we obtain \( f(S_2) \leq \frac{f(S)}{2} \).

By these inequalities, the cache placement is feasible if and only if \( f(S_1) = f(S_2) = \frac{f(S)}{2} \). This means that finding a feasible solution of the cache placement problem defined as above is equivalent to solving the original PARTITION problem, which completes the proof.

**Remark 3.1.** The above proof of Theorem 3.4 shows that, for the cache placement problem with link capacities, even finding a feasible solution is NP-hard. This means that we cannot expect the existence of an efficient \( \alpha \)-approximation algorithm for any constant \( \alpha \).

By using the same argument as above, we can easily see that the cache placement problem with link capacity is also NP-hard when an original file of each item is placed at some node.

### 4. Reducing to Network Flows

In this section, we aim to give a more efficient algorithm for the cache placement problem (2.1) with a practical assumption. As mentioned in Section 1, making many copies of an item might cause some problems in terms of security and maintenance. With this background, it is natural to assume that we have an upper bound on the number of copies of an item. In this section, we consider the following setting.

- For each item \( i \), there exists a fixed node \( u_i \) which has data \( i \), i.e., the original data of \( i \) is in \( u_i \).

- For each item \( i \), we can choose at most one node \( w_i(\neq u_i) \) which has a copy of \( i \).

The other constraints and the objective function are the same as the cache placement problem in Section 2. That is, each core node \( u \) in \( U \) can store \( q(u) \) items, and the objective is to minimize

\[
\sum_{i \in I} \sum_{v \in V} d(v, i) \min\{\text{dist}(v, u_i), \text{dist}(v, w_i)\},
\]
because a client \(v\) accesses \(u_i\) or \(w_i\) to obtain item \(i\). We call this problem the \textit{cache placement problem with at most one copy}, and in what follows, we solve this problem by using minimum-cost flow algorithms.

Suppose we are given an instance of the cache placement problem with at most one copy. That is, we are given a graph \(G = (V, E)\), cache size \(q(u)\), demand \(d(v, i)\), distance \(\text{dist}(u, v)\), node \(u_i\) having the original file of \(i\). We define \(q'(u) := q(u) - |\{i \in I \mid u_i = u\}|\) for \(u \in U\). This means that node \(u\) can store copies of at most \(q'(u)\) items (other than original file). Also define

\[
g(w, i) := \sum_{v \in V} d(v, i) \max\{0, \text{dist}(v, u_i) - \text{dist}(v, w)\}
\]

for each \(w \in U\) and \(i \in I\). Note that if we place a copy of item \(i\) in \(w\), then it reduces the cost by \(g(w, i)\).

We construct a new network (directed graph) \(N\) as follows. The node set consists of four layers \(V_1 := \{s\}\), \(V_2 := \{z_i \mid i \in I\}\), \(V_3 := U \cup \{u^*\}\), and \(V_4 := \{t\}\). The arc set consists of all arcs from \(V_j\) to \(V_{j+1}\) for \(j = 1, 2, 3\). We define a capacity constraint (lower bound \(c(e)\) and upper bound \(\bar{c}(e)\)) and a cost \(\gamma(e)\) of each arc \(e\) as follows (see Figure 4).

1. For an arc \(e\) from \(s \in V_1\) to \(z_i \in V_2\), define \(c(e) = \bar{c}(e) = 1\) and \(\gamma(e) = 0\).
2. For an arc \(e\) from \(z_i \in V_2\) to \(w \in V_3\), define \(c(e) = 0\), \(\bar{c}(e) = 1\), and \(\gamma(e) = -g(w, i)\) if \(w \in U\) and \(\gamma(e) = 0\) if \(w = u^*\).
3. For an arc \(e\) from \(w \in V_3\) to \(t \in V_4\), define \(c(e) = 0\), \(\bar{c}(e) = q'(w)\) if \(w \in U\) and \(\bar{c}(e) = +\infty\) if \(w = u^*\), and \(\gamma(e) = 0\).

Figure 4: Reduction to the minimum-cost flow problem

We now show that an integral feasible solution of this minimum-cost flow problem corresponds to a feasible cache placement. Consider an integral feasible flow of this problem. For any \(z_i \in V_2\), since a flow value through \(e = (s, z_i)\) is exactly one, there exists exactly one element \(w \in V_3\) such that one unit of flow is sent from \(z_i\) to \(w\). In such a case, we place a copy of item \(i\) to \(w\) if \(w \in U\) and we do not make any copy of item \(i\) if \(w = u^*\). By this construction, we can make a correspondence between an integral feasible flow and a feasible cache placement. We can easily see that the cost of the flow is equal to the minus of the total cost reduced by the corresponding cache placement.

Since each capacity constraint is integral, the above minimum-cost flow problem has an integral optimum solution, and it can be found in polynomial time (see e.g., [4, 26]). Therefore, by solving the minimum-cost flow problem, we can find an optimal cache placement in polynomial time.
Theorem 4.1. There is a polynomial-time algorithm for the cache placement problem with at most one copy.

5. Computational Experiments

To evaluate the performance of our algorithms for the cache placement problem, we have conducted simulation on SINET 4. We implemented two algorithms: the one to solve the problems (2.1) and (2.3) with an IP solver, and the other one using a minimum-cost flow algorithm as described in Section 4. An algorithm in Section 3.2 is not implemented because it is expected to be slower than using an IP solver when applying to SINET.

5.1. Setting details

SINET 4 consists of 11 core nodes and more than 100 edge nodes. Edge nodes are partitioned into some groups, and all the edge nodes in a group are linked to the same core node in SINET 4. There is no link between edge nodes in different groups. For simplicity, we ignore other small nodes linked to edge nodes, and we model the network as the graph only with the core nodes and the edge nodes. We also merge some nodes in the same place. The resulting graph is denoted by $G = (V, E)$, which has 8 core nodes, 127 edge nodes, and 211 links. We denote $V = V_{\text{core}} \cup V_{\text{edge}}$, where $V_{\text{core}}$ (resp. $V_{\text{edge}}$) is the set of all core nodes (resp. edge nodes). A link between core nodes is called a core link, and a link between a core node and an edge node is an edge link. In our problem setting, we assume that cache files are placed only at a core node $u$, and their cache size $q(u)$ will be determined in each experiment. We assume that each item is of the same size, and hence the size of a cache memory is regarded as the number of items which can be stored in a cache.

We set the length $c(e)$ of each core link $e$ as 5, and the length $c(e)$ of each edge link $e$ as 1. We assume that when we access item $i$ from node $v$, we access the nearest node having a copy of $i$ via the shortest path (with respect to the length $c$).

We set the demand of each node as follows. In SINET 4, we measure at each time the amount of the traffic on each link, and the total amount of the traffic entering (or leaving) each node. We define $d(v)$ at an edge node $v \in V_{\text{edge}}$ as the total amount of the traffic entering it in a certain period. Figure 5 visualizes the distribution of $d(v)$’s, where the area of each circle indicates the total amount of demands through the corresponding core node. We suppose that the demand $d(v, i)$ of item $i$ at node $v$ is based on a probabilistic distribution. That is, $d(v, i)$ is defined as the product of the total demand $d(v)$ at $v$ and a demand distribution $D(i)$ with $\sum_{i \in I} D(i) = 1$, i.e., $d(v, i) = d(v)D(i)$. In our experiments, we adopt two kinds of demand distributions: the uniform distribution and a distribution determined by Zipf’s law, i.e., the frequency of the $p$-th most popular item is proportional to $1/p^s$, where $s$ is a parameter. More precisely, the probability of the $p$-th item in Zipf’s law is

$$1/p^s \sum_{j=1}^{k} 1/j^s. \quad (5.1)$$

We set $s = 1$ unless otherwise specified.

Experiments were performed on a Windows PC with Intel Core i7 CPU 2.40GHz and 16.0GB memory. Integer optimization problems were solved by IBM ILOG CPLEX 12.5 implemented with Microsoft Visual C++ 2010 Express, and the network-flow-based algorithm was implemented with gcc4.5.3 (Cygwin).

*The new version SINET 5 has been launched on April 2016.*
5.2. Solving integer optimization exactly

In this subsection, we demonstrate experimental results when solving the integer optimization problems in Section 2. The purpose of the experiments is to confirm the effectiveness of optimal cache placement and to analyze optimal solutions.

We consider two cases: the one when at least one cache file, i.e., an original file, has been already placed for each item and the one when no original file is placed at the beginning. For the former case, we generate randomly an initial allocation of original files 100 times, and take the average of their results. We assume that an original file is not contained in cache, that is, a cache size $q(u)$ means the remaining cache size after placing original files.

Figures 6 and 7 show the optimal values of the problem (2.1) when we have 10 items. The vertical axis indicates the optimal transmission cost divided by the total amount of all demands to normalize. We can see from the figures that if a large cache size is allowed, then the optimal transmission cost is significantly decreased. Especially, when we compare the cases with cache size 3 and with no caches under the Zipf’s law distribution, the cost is improved by more than 50% when having original files, and about 20% when having no original files. We can also observe that cache allocation is more useful when a probabilistic distribution is biased (i.e., one based on Zipf’s law).

We also analyzed relationship between demand distributions and optimal solutions by changing the parameter $s$ of (5.1) from 0 to 1. Figure 8 summarizes the result when we have 10 items and cache size 3. Remark that the case when $s = 0$ corresponds to the uniform distribution. Interestingly, one can see that optimal solutions are positively correlated with the demand distribution. That is, frequent items are stored in many places, which are distributed evenly. Note that the left-most column when $s = 1.0$ attains the number of core nodes, because it is meaningless to place an item twice at the same place.

In addition, we observed the significance of link capacity constraints. To do this, we measured the amount of transmitted data of each link (distinguishing directions) in optimal
Figure 6: Cache sizes and optimal values with initial allocation

Figure 7: Cache sizes and optimal values without initial allocation

Figure 8: Demand distributions and placed numbers of items
solutions under the Zipf’s law distribution. Figure 9 shows the data amount of links when we have no caches and when we have link capacity constraints with cache size 3. We here set a link capacity to be 0.05D for each core link, where D is the total amount of demand data in the whole network. The figure illustrates the top 6 links of the former case, which contain the top 5 links of the latter case. The labels of the links are corresponding to those in Figure 5. Note that, for all 100 random instances, our algorithm finds a feasible solution, which implies that we can reallocate caches to reduce bandwidth in most cases. We can see from Figure 9 that data was congested around Tokyo, but caching can reduce the maximum amount of transmitted data by about 80%. We also remark that adding a link capacity constraint did not change the optimal transmission cost. In fact, the optimal value gets worse only by 1.4%.

Figure 9: Amount of transmission through each link

5.3. Network-flow-based algorithm
We executed our network-flow-based algorithm in Section 4 for SINET with a similar setting. We assume that, for each item, an original file has already been allocated at some core node, and that each item can choose at most one node to have its copy. We generate randomly an initial allocation of original files 10 times, and take the average of their results.

Figure 10 shows the optimal values when the cache size is changed. We can see that since we can place only one copy for each item, we cannot improve the transmission cost better than the value obtained in Section 5.2. For example, when the cache size is 3, the optimal value in this case is about half again the size of the value in Figure 6. However, we still have significant improvement compared to the case without caches. Moreover, when the cache size is one, it achieves almost the same value as in Figure 6. We also measured the amount of transmitted data of each link in optimal solutions similarly to Figure 9 (without edge capacity), which is shown in Figure 11. (Note that in this figure, we also show the same values for “no cache” setting in Figure 9.) When the cache size is 3, the maximum amount of transmitted data is achieved at link 1. Although the maximum amount becomes smaller, it is worse than that in Section 5.2.

5.4. Heuristic using network-flow-based algorithm
Our network-flow-based algorithm can be used to design an efficient heuristic to solve the original cache placement problem. In our heuristic, we repeatedly solve an instance where each item can be put at most once. More specifically, assuming that the caches already assigned are fixed, we consider an instance where we can put at no more than one cache for each item, and find a solution using our network-flow-based algorithm in Section 4. By
repeating this procedure until there is no available place, we find a solution for the original instance. Note that, after $t$ iterations, we have a solution that each item is put at no more than $t$ places.

To evaluate our heuristic, we compare the following algorithms.

1. **Random**: choose a solution uniformly at random
2. **Greedy**: compute the increase $g(w, i)$, defined in (4.1), when item $i \in I$ is put at node $w \in U$, and choose items with maximum $g(w, i)$ in a greedy way.
3. **IterativeFlow($k$)**: repeatedly solve our network-flow-based algorithm with flow limit $k$.

In our heuristic, we solve a minimum-cost flow problem repeatedly, but finding an optimal flow in each iteration might not be good in the sense of the quality of the final solution. Thus we set a parameter $k$, which is the limit of flow, and we find a minimum-cost flow of value $k$ in each iteration.

Table 1 shows the solution quality of each algorithm with different parameters. We generate 20 random instances for each setting, and take the average of all the results, where the values are the ratio of the output to the optimal value. Note that we conduct experiments of **Random** 100 times for each instance and take the average value.

We can observe that **IterativeFlow** always returns the best solution. When the distribution is uniform (i.e., $s = 1$), all the algorithms **IterativeFlow($k$)** with $k = 1, |I|/2, |I|$ return near optimal solutions. In fact, the ratio to the optimal value is within at most 1%. However, the problem becomes more difficult when the distribution is biased (i.e., $s$ becomes large). In this case, all the algorithms are of less performance, but **IterativeFlow(1)** still returns a solution whose value is at most 1.4 times the optimal value. **Random** is always the worst among all the algorithms. **Greedy** returns a relatively good solution, but it becomes less efficient when the capacity $q(u)$ for each node $u$ is small. This is because some core node $w$ has large $g(w, i)$ for many items $i$, but we cannot assign two items at one place. On the
Table 1: Performance of heuristic algorithms (The best value at each row is indicated by boldface. Numbers less than 1.03 are red (symbolized by *), and numbers less than 1.05 are blue (symbolized by **)).

| | q | s | Random | Greedy | IterativeFlow(1) | IterativeFlow(|I|/2) | IterativeFlow(|I|) |
|---|---|---|---|---|---|---|---|
| 10 | 0.0 | 1.1487 | 1.1973 | 1.0085* | 1.0294* | 1.0007* | 1.1246 | 1.1875 | 1.0033* | 1.0080* | 1.0004* | 1.0707 | 1.1430 | 1.0000* | 1.0023* | 1.0015* | 1.0752 | 1.1205 |
| | 0.25 | 1.1700 | 1.1890 | 1.0176* | 1.0232* | 1.0017* | 1.1359 | 1.1603 | 1.0246* | 1.0203* | 1.0206* | 1.0203 | 1.0206 | 1.0203 | 1.0206 | 1.0217* | 1.0241* | 1.0491* | 1.0491* | 1.0491* | 1.0491* |
| | 0.5 | 1.2238 | 1.1644 | 1.0149* | 1.0165* | 1.0165* | 1.2080 | 1.1472 | 1.0315** | 1.0307** | 1.0491** | 1.0351 | 1.0351 | 1.0351 | 1.0351 | 1.0382** | 1.0382** | 1.1021 | 1.1021 | 1.1021 | 1.1021 |
| 20 | 0.0 | 1.1260 | 1.1753 | 1.0068* | 1.0047* | 1.0013* | 1.1260 | 1.1753 | 1.0068* | 1.0047* | 1.0013* | 1.1260 | 1.1753 | 1.0068* | 1.0047* | 1.0013* | 1.1260 | 1.1753 | 1.0068* | 1.0047* | 1.0013* |
| | 0.25 | 1.1693 | 1.1760 | 1.0263* | 1.0093* | 1.0073* | 1.2359 | 1.1409 | 1.0295* | 1.0223* | 1.0288* | 1.2359 | 1.1409 | 1.0295* | 1.0223* | 1.0288* | 1.2359 | 1.1409 | 1.0295* | 1.0223* | 1.0288* |
| | 0.5 | 1.2359 | 1.1409 | 1.0295* | 1.0223* | 1.0288* | 1.4365 | 1.1296 | 1.0346** | 1.0503 | 1.0770 | 1.5592 | 1.1127 | 1.0296* | 1.1034 | 1.1533 | 1.5592 | 1.1127 | 1.0296* | 1.1034 | 1.1533 |
| 6 | 0.0 | 1.0665 | 1.1108 | 1.0000* | 1.0000* | 1.0000* | 1.0665 | 1.1108 | 1.0000* | 1.0000* | 1.0000* | 1.0665 | 1.1108 | 1.0000* | 1.0000* | 1.0000* | 1.0665 | 1.1108 | 1.0000* | 1.0000* | 1.0000* |
| | 0.25 | 1.1392 | 1.1522 | 1.0297* | 1.0260* | 1.0260* | 1.2761 | 1.1378 | 1.0359** | 1.0348** | 1.0654 | 1.2761 | 1.1378 | 1.0359** | 1.0348** | 1.0654 | 1.2761 | 1.1378 | 1.0359** | 1.0348** | 1.0654 |
| | 0.75 | 1.4784 | 1.1330 | 1.0391** | 1.0629 | 1.1393 | 1.7343 | 1.1118 | 1.0357** | 1.1044 | 1.2512 | 1.7343 | 1.1118 | 1.0357** | 1.1044 | 1.2512 |

Other hand, since each iteration of IterativeFlow assigns items at once avoiding such conflicts, IterativeFlow returns a better solution when \( q(u) \) is smaller for each node \( u \).

Finally, we observe that, when we have more items, IterativeFlow does not return a good solution. In particular, IterativeFlow(|I|) returns a worse solution than Greedy when \(|I| = 20\) and \(s = 1.0\). This is because we assign \(|I|\) different items in each iteration of IterativeFlow, and then we cannot assign an item with much improvement in the next iteration. The difference is made clearer when the distribution is biased. On the other hand, IterativeFlow(1) is still of good performance, since we re-calculate the increase \(g(w, i)\) when we put item \(i\) at node \(w\) every time we find a flow of value 1.

6. Conclusion
In this paper, we proposed an integer optimization model to allocate caches effectively in SINET. We designed a dynamic programming algorithm to solve the model when the treewidth is small, and moreover, we devised a polynomial-time algorithm under a practical assumption by reducing to the minimum-cost flow problem. We also considered imposing an additional constraint to keep small bandwidth. We showed that this constraint makes the

other hand, since each iteration of IterativeFlow assigns items at once avoiding such conflicts, IterativeFlow returns a better solution when \( q(u) \) is smaller for each node \( u \).

Finally, we observe that, when we have more items, IterativeFlow does not return a good solution. In particular, IterativeFlow(|I|) returns a worse solution than Greedy when \(|I| = 20\) and \(s = 1.0\). This is because we assign \(|I|\) different items in each iteration of IterativeFlow, and then we cannot assign an item with much improvement in the next iteration. The difference is made clearer when the distribution is biased. On the other hand, IterativeFlow(1) is still of good performance, since we re-calculate the increase \(g(w, i)\) when we put item \(i\) at node \(w\) every time we find a flow of value 1.

6. Conclusion
In this paper, we proposed an integer optimization model to allocate caches effectively in SINET. We designed a dynamic programming algorithm to solve the model when the treewidth is small, and moreover, we devised a polynomial-time algorithm under a practical assumption by reducing to the minimum-cost flow problem. We also considered imposing an additional constraint to keep small bandwidth. We showed that this constraint makes the
problem too hard to find a feasible solution. Our numerical experiments on the real SINET network confirm that our algorithms can solve the cache placement problem efficiently in practice. In particular, when a demand distribution follows Zipf’s law, allocating caches in SINET is shown to be quite useful to reduce the transmission cost. One of our future work is an online management of cache placement. In reality, demands in a network are varying day by day, and thus we need to change cache placement accordingly. It would be interesting to design an optimal mechanism to modify a cache allocation to adapt current demands.

Acknowledgement
The authors thank the referees for their valuable comments. This work was supported by JST ERATO Grant Number JPMJER1201, Japan. The second author is supported by JSPS KAKENHI Grant Number JP17K00028. The fifth (sixth) author is (was) supported by JSPS Research Fellowship for Young Scientists.

References
[1] http://w4a.sinet.ad.jp/Network/domestic2.jpg (accessed on November 8, 2017).
[2] SINET4 brochure: https://www.sinet.ad.jp/wp-content/uploads/2016/03/SINET4-2014e.pdf (accessed on November 8, 2017).
[3] SINET4 webpage: http://w4a.sinet.ad.jp/index_en.html?lang=english (accessed on November 8, 2017).
[4] R.K. Ahuja, T.L. Magnanti, and J.B. Orlin: Network Flows: Theory, Algorithms, and Applications (Prentice hall, 1993).
[5] S. Arnborg and A. Proskurowski: Linear time algorithms for NP-hard problems restricted to partial \( k \)-trees. *Discrete Applied Mathematics*, 23 (1989), 11–24.
[6] V. Arya, N. Garg, R. Khandekar, A. Meyerson, K. Munagala, and V. Pandit: Local search heuristics for \( k \)-median and facility location problems. *SIAM Journal on Computing*, 13 (2004), 544–562.
[7] I.D. Baev, R. Rajaraman, and C. Swamy: Approximation algorithms for data placement problems. *SIAM Journal on Computing*, 38 (2008), 1411–1429.
[8] H.L. Bodlaender: A linear-time algorithm for finding tree-decomposition of small treewidth. *SIAM Journal on Computing*, 25 (1996), 1305–1317.
[9] S.C. Borst, V. Gupta, and A. Walid: Distributed caching algorithms for content distribution networks. In *Proceedings of 29th IEEE International Conference on Computer Communications* (2010), 1478–1486.
[10] M. Charikar, S. Guha, É. Tardos, and D. Shmoys: A constant-factor approximation algorithm for the \( k \)-median problem. *Journal of Computer and System Sciences*, 65 (2002), 129–149.
[11] H. Che, Z. Wang, and Y. Tung: Analysis and design of hierarchical Web caching systems. In *Proceedings of 20th IEEE International Conference on Computer Communications* (2001), 1416–1424.
[12] V. Jacobson, D.K. Smetters, J.D. Thornton, M.F. Plass, N. Briggs, and R. Braynard: Networking named content. *Communications of the ACM*, 55 (2012), 117–124.
[13] K. Jain, M. Mahdian, and A. Saberi: A new greedy approach for facility location problems. In *Proceedings of the 34th ACM Symposium on Theory of Computing* (2002), 731–740.
[14] J. Kangasharju, J.W. Roberts, and K.W. Ross: Object replication strategies in content distribution networks. *Computer Communications*, **25-4** (2002), 376–383.

[15] R.M. Karp: On the computational complexity of combinatorial problems. *Networks*, **5** (1975), 45–68.

[16] M.R. Korupolu, C.G. Plaxton, and R. Rajaraman: Placement algorithms for hierarchical cooperative caching. *Journal of Algorithms*, **38** (2001), 260–302.

[17] R. Krishnaswamy, A. Kumar, V. Nagarajan, Y. Sabharwal, and B. Saha: The matroid median problem. In *Proceedings of the 22nd Annual ACM-SIAM Symposium on Discrete Algorithms* (2011), 1117–1130.

[18] S. Li: A 1.488 approximation algorithm for the uncapacitated facility location problem. *Information and Computation*, **222** (2013), 45–58.

[19] M. Mangili, F. Martignon, and A. Capone: A comparative study of content-centric and content-distribution networks: Performance and bounds. In *Proceedings of IEEE GLOBECOM* (2013), 1403–1409.

[20] M.C. Mihaescu, D.D. Burdescu, M. Mocanu, and C.M. Ionascu: Load balancing procedure for building distributed e-learning systems. In *Proceedings of the 3rd International Conference on Mobile, Hybrid, and On-line Learning* (2011), 82–87.

[21] J. Ni and D.H.K. Tsang: Large-scale cooperative caching and application-level multicast in multimedia content delivery networks. *IEEE Communications Magazine*, **43** (2005), 98–105.

[22] L. Perkovic and B. Reed: An improved algorithm for finding tree decompositions of small width. *International Journal of Foundations of Computer Science*, **11** (2000), 81–85.

[23] N. Robertson and P.D. Seymour: Graph minors. III. Planar tree-width. *Journal of Combinatorial Theory, Series B*, **36** (1984), 49–64.

[24] N. Robertson and P.D. Seymour: Graph minors XIII. The disjoint paths problems. *Journal of Combinatorial Theory, Series B*, **63** (1995), 65–110.

[25] N. Robertson and P.D. Seymour: Graph minors. XX. Wagner’s conjecture. *Journal of Combinatorial Theory, Series B*, **92** (2004), 325–357.

[26] A. Schrijver: *Combinatorial Optimization – Polyhedra and Efficiency* (Springer-Verlag, 2003).

[27] K. Slavicek and V. Novak: Fiber optics transport infrastructure of cesnet backbone. In *Proceedings of the 6th WSEAS international conference on Applied computer science* (2006), 323–328.

[28] C. Swamy: Improved approximation algorithms for matroid and knapsack median problems and applications. In *Proceedings of the 17th International Workshop on Approximation Algorithms for Combinatorial Optimization Problems* (2014), 403–418.

[29] A. Tamir: An $O(pn^2)$ algorithm for the $p$-median and related problems on tree graphs. *Operations Research Letters*, **19** (1996), 59–64.

[30] S. Urushidani, M. Aoki, K. Fukuda, S. Abe, M. Nakamura, M. Koibuchi, Y. Ji, and S. Yamada: Highly available network design and resource management of SINET4. *Telecommunication Systems*, **56** (2014), 33–47.

[31] H. Yuan and P. Crowley: Performance measurement of name-centric content distribution methods. In *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems* (2011), 223–224.
Yu Yokoi
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku,
Tokyo 101-8430, Japan
E-mail: yokoi@nii.ac.jp