Network File Storage With Graceful Performance Degradation

ANXIAO (ANDREW) JIANG and JEHOSHUA BRUCK
California Institute of Technology

A file storage scheme is proposed for networks containing heterogeneous clients. In the scheme, the performance measured by file-retrieval delays degrades gracefully under increasingly serious faulty circumstances. The scheme combines coding with storage for better performance. The problem is NP-hard for general networks; and this article focuses on tree networks with asymmetric edges between adjacent nodes. A polynomial-time memory-allocation algorithm is presented, which determines how much data to store on each node, with the objective of minimizing the total amount of data stored in the network. Then a polynomial-time data-interleaving algorithm is used to determine which data to store on each node for satisfying the quality-of-service requirements in the scheme. By combining the memory-allocation algorithm with the data-interleaving algorithm, an optimal solution to realize the file storage scheme in tree networks is established.

Categories and Subject Descriptors: C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*Distributed applications; Distributed databases*; C.4 [**Performance of Systems**]: *Reliability, availability, and serviceability*; E.4 [**Coding and Information Theory**]: *Error control codes*; E.5 [**Files**]: *Backup / recovery*; F.2.2 [**Analysis of Algorithms and Problem-Complexity**]: Nonnumerical-Algorithms and Problems—*Computations on discrete structures; Routing and layout*; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph algorithms; Network problems; Trees*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models*

General Terms: Algorithms, Performance, Reliability, Theory

Additional Key Words and Phrases: Domination, file assignment, interleaving, memory allocation, fault tolerance

1. INTRODUCTION

A file shared by many distributed clients can be replicated in the network to improve performance, and the file can be stored in the form of an error-correcting code. Let's use (N, ε) *code* to denote an error-correcting code that consists of N symbols and can correct ε erasures—in other words, any $N - \varepsilon$

This work was supported in part by the Lee Center for Advanced Networking at the California Institute of Technology, and by NSF grant CCR-TC-0208975.

Authors' address: Electrical Engineering Department, California Institute of Technology, MS 136-93, Pasadena, CA 91125; email: {jax,bruck}@paradise.caltech.edu.

symbols can be used for decoding the codeword. Given a file, we can encode it with an (N, ε) code, and distributively store replicas of the N symbols of the codeword in the network. Then each client can recover the file by retrieving any $N - \varepsilon$ different symbols.

The most common practice of file storage, where every node of the network either stores the entire file in its original form or none of it, is a topic that has been studied in depth [Dowdy and Foster 1982]. It includes median or center types of schemes that minimize the average or maximum file-access cost [Hochbaum and Shmoys 1985; Kalpakis et al. 2001], dynamic replication schemes based on estimated temporal data-access statistics (e.g., caching [Wang 1999]), on-line algorithms that optimize the file-access performance against the worst future events [Borodin and El-Yaniv 1998], etc. In those schemes, the file can be seen as encoded with a $(1, 0)$ code, so they are a special case of the more general file-storage model where files are stored in the form of error-correcting codes. There also exist schemes using *file segmentation* [Mahmoud and Riordan 1976], where a file is split into chunks and the chunks are stored distributively, which can be seen as using a $(k, 0)$ code (for some integer k). Error-correcting codes have played a more important role in disk-storage systems and server clusters—such as RAID [Patterson et al. 1988] and DPSS [Malluhi and Johnston 1998]—where files are stored using nontrivial error-correcting codes, but there the concept of network is not significant. Works that study the general problem of combining network file storage with error-correcting codes include the important article by Naor and Roth [1995]—which studied how to store a file using error-correcting codes in a network such that every node can recover the file by accessing only the codeword symbols on itself and its neighbors, with the objective of minimizing the total amount of data stored—and a few other results [Jiang and Bruck 2003b, 2005; Jiang et al. 2004]; however, other than those, research in this field has been very limited.

Error-correcting code is a more general way to express a file than the file itself. Therefore, it brings us the flexibility of finding file-storage solutions with better performance.

In this article, we study file storage in networks containing heterogeneous clients—clients that have different quality-of-service requirements on file retrieval. We model a network as a directed graph $G = (V, E)$, and use (u, v) to denote a directed edge from vertex u to vertex v. Each edge $(u, v) \in E$ has a positive length $l(u, v)$. We use $d(u \to v)$ to denote the length of the shortest directed path from $u \in V$ to $v \in V$, and call it the *distance from u to v*. For a vertex $v \in V$ and a real number r, we define $\mathcal{N}(v, r)$ as the set of vertices whose distance to v is less than or equal to r, namely, $\mathcal{N}(v, r) = \{u | u \in V, d(u \to v) \leq r\}$. We encode a file with an (N, ε) code, and store replicas of the N codeword symbols on the vertices of the graph. Every vertex is a client that requests the file; at the same time, it can be used to store some codeword symbols. We use $W_{\max}(v)$ to denote the maximum number of codeword symbols that can be stored on vertex $v \in V$, and call it the *memory capacity of* v. If a vertex v retrieves codeword symbols from a set $S \subseteq V$ of vertices, then we call $\max_{u \in S} d(u \to v)$ the *file-retrieval delay* of v. (So here the length of a path is interpreted as the delay of transmitting data over that path.)

We allow every vertex to specify a delay that it can tolerate for retrieving $N - \varepsilon$ different codeword symbols for its file reconstruction. If some of the stored data becomes inaccessible (e.g., because of data loss or processors' being busy), then a vertex needs to retrieve codeword symbols from a larger area. It is desirable that the number of distinct codeword symbols within a distance from a vertex grows steadily when that distance increases—so that the file-retrieval delay will degrade gracefully when more and more symbols become inaccessible. We let each vertex specify the number of distinct codeword symbols that should exist within each specified distance, and we allow different vertices to have different such requirements. As a result, we get a file-storage scheme accommodating the varied quality-of-service requirements of clients, which has graceful performance degradation under increasingly serious faulty circumstances.

The problem studied in this article is formally defined as follows.

Definition 1.1. *The File Storage Problem*

Instance: A directed graph $G = (V, E)$, and a codeword of N symbols. Every edge $(u, v) \in E$ has a positive length $l(u, v)$. ($l(u, v)$ is a real number.) Every vertex $v \in V$ is associated with a set $R(v) = \{(r_i(v), k_i(v)) | 1 \leq i \leq n_v\}$, which is called the *requirement set of* v. Each vertex $v \in V$ is also associated with a nonnegative integer $W_{\max}(v)$, which is called the *memory capacity of* v.

Question: How to assign $w(v)$ codeword symbols to each vertex $v \in V$, such that for every vertex $u \in V$ and for $1 \leq i \leq n_u$, the vertices in the set $\mathcal{N}(u, r_i(u))$ together have at least $k_i(u)$ distinct codeword symbols? Here $w(v) \leq W_{\max}(v)$ for all $v \in V$. $w(v)$ is called the *memory size of* v. A feasible solution to this problem that minimizes the total number of codeword symbols stored in the graph, $\sum_{v \in V} w(v)$, is called an *optimal solution*.

Comments: Each element in a *requirement set* $R(v)$ is a pair of numbers, written in the form (r, k). $r_i(v)$ is a nonnegative real number. $k_i(v)$, n_v, $W_{\max}(v)$, and $w(v)$ are all nonnegative integers. n_v denotes the number of requirements that v has.

The file storage problem defined above is NP-hard for general graphs, because the NP-complete dominating set problem [Garey and Johnson 1979] can be reduced to it. In this article, we study the case where the graph $G = (V, E)$ is a tree. We assume G has asymmetric edges, which means that, for any two adjacent vertices, the two directed edges of opposite directions between them do not necessarily have the same length. Below is an example of such a file storage problem.

Example 1.1. A tree G with asymmetric edges is shown in Figure 1, where the number beside each edge is its length. The parameters N, $R(v)$, and $W_{\max}(v)$ (for every vertex v) are as shown. (So here $n_{v_1} = 2$, $n_{v_2} = n_{v_3} = \cdots = n_{v_6} = 1$.)

Let's use integers $1, 2, \ldots, 12$ to denote the 12 codeword symbols. Then one feasible solution is as follows: assign $w(v_1) = 3$ symbols—$\{1, 2, 3\}$—to v_1, assign $w(v_2) = 3$ symbols—$\{9, 10, 11\}$—to v_2, assign $w(v_3) = 0$ symbol to v_3, assign $w(v_4) = 5$ symbols—$\{4, 5, 6, 7, 8\}$—to v_4, assign $w(v_5) = 7$ symbols—$\{1, 2, 3, 9, 10, 11, 12\}$—to v_5, assign $w(v_6) = 0$ symbol to v_6. We claim without

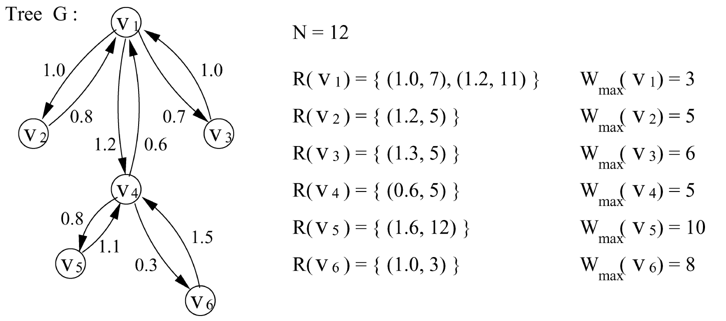

Fig. 1. An example of the file storage problem.

proof that that solution is *optimal*, because it minimizes the value $\sum_{v \in V} w(v)$; readers can verify that the claim is true.

The example here is a simple one. In general, a vertex can have much more than one or two requirements.

Trees are often used as embedded networks or backbone networks in real systems. In those networks, the cost (such as delay) of transmitting data from one node to another is often not the same as the cost of transmitting data in the opposite direction. Trees with asymmetric edges take that fact into consideration. They include undirected trees as a special case.

Finding a solution to the file storage problem has two steps: deciding how many codeword symbols to assign to each vertex, which we call *memory allocation*, and deciding which codeword symbol to assign to each vertex, which we call *data interleaving*. If G is a general graph, these two steps usually depend on each other. However, we will show that when G is a tree, *memory allocation* and *data interleaving* can be solved separately.

The rest of the article is organized as follows. Sections 2 and 3, respectively, present a memory-allocation algorithm (Algorithm 2.1) and a data-interleaving algorithm (Algorithm 3.1), both of polynomial time complexity. The combination of those two algorithms yields an optimal solution to the file storage problem, and that result is shown in Section 4. Section 5 presents concluding remarks. Appendix A presents the pseudocode of Algorithm 2.1, Appendix B presents the proof of and complexity analysis for Algorithm 2.1, Appendix C presents the algorithm for the memory-allocation problem without an upper bound for memory sizes (Algorithm C.1), and Appendix D discusses the complexity of the data-interleaving algorithm (Algorithm 3.1).

2. MEMORY ALLOCATION

2.1 Definition of the Problem

We define the *memory-allocation problem* as follows.

Definition 2.1. *The Memory-Allocation Problem*.

Instance: A tree $G = (V, E)$ with asymmetric edges, and a positive integer N. Every edge $(u, v) \in E$ has a positive length $l(u, v)$. Every vertex $v \in V$ is associated with a set $R(v) = \{(r_i(v), k_i(v)) | 1 \le i \le n_v\}$, which is called the *requirement set of* v. Each vertex $v \in V$ is also associated with a nonnegative integer $W_{\max}(v)$, which is called the *memory capacity of* v, and a nonnegative integer $W_{\min}(v)$, which is called the *memory floor of* v. (Here $W_{\min}(v) \leq W_{\max}(v)$.)

Question: How to associate an integer $w(v)$ with each vertex $v \in V$, such that for every vertex $u \in V$ and for $1 \leq i \leq n_u$, $\sum_{v \in \mathcal{N}(u, r_i(u))} w(v) \geq k_i(u)$? Here $W_{\min}(v) \leq w(v) \leq W_{\max}(v)$ for all $v \in V$. $w(v)$ is called the *memory size of* v. A feasible solution to this problem that minimizes the value $\sum_{v \in V} w(v)$ is called an *optimal solution*.

Comments: All the parameters above—except the new parameter $W_{\min}(v)$—have the same meaning as in the file storage problem (Definition 1.1). So we omit defining their allowed ranges of values.

The memory-allocation problem has one generalization compared to the file storage problem—an integer $W_{\min}(v)$, instead of the constant 0, is set to be the lower bound for the memory size of v. Other than that, the memory allocation problem is a simplification of the file storage problem—instead of requiring that there are at least $k_i(v)$ *distinct* codeword symbols stored on the vertices in the set $\mathcal{N}(v, r_i(v))$, the memory allocation problem just requires at least $k_i(v)$ codeword symbols (whether they are the same or not) to be stored there. Clearly, in the case where $W_{\min}(v) = 0$ for all $v \in V$, if an optimal solution to the memory allocation problem assigns the integer $w(v)$ to vertex v, then $\sum_{v \in V} w(v)$ is a lower bound for the total number of codeword symbols stored in the tree in any feasible solution to the file storage problem. In later sections, we will show that in fact, storing $\sum_{v \in V} w(v)$ codeword symbols is also sufficient.

We assume in the rest of the article that for every vertex $u \in V$ and for $1 \leq i \leq n_u$, $\sum_{v \in \mathcal{N}(u, r_i(u))} W_{\max}(v) \geq k_i(u)$, because that is the necessary and sufficient condition for there to exist a solution to the memory allocation problem.

2.2 Memory-Allocation Algorithm

We see one of the vertices of the tree G as its *root*, and denote it by v_{root}. For any two vertices v_1 and v_2, we say "v_1 is a descendant of v_2" or "v_2 is an ancestor of v_1" if $v_2 \neq v_1$ and v_2 is on the shortest path from the root to v_1. We say "v_1 is a child of v_2" or "v_2 is the parent of v_1" if v_1 and v_2 are adjacent and v_1 is a descendant of v_2. For any vertex $v \in V$, we use $Des(v)$ to denote the set of descendants of v.

For any set S, we use $|S|$ to denote its cardinality. For any two sets S and T, $S - T$ denotes the set of elements that are in S but not in T. For any two variables a and b, $a \leftarrow b$ means to make a be equal to b (in other words, it means to assign the value of b to a).

We present below a memory-allocation algorithm that uses the technique of searching the tree from its leaves toward its root. Similar techniques have been used in several articles [Kariv and Hakimi 1979; Slater 1976] to solve the domination problem.

Definition 2.2. *An Optimal Memory Basis*.

A set $\{w(v) | v \in V\}$ is called an *optimal memory basis* if there exists an optimal solution to the memory allocation problem which assigns the integer $w_{opt}(v)$ to every vertex $v \in V$, such that for every vertex $v \in V$, $W_{\min}(v) \leq w(v) \leq w_{opt}(v)$.

The following lemma shows how, given an optimal memory basis $\{w_1(v)|v \in V\}$, one can derive a new optimal memory basis $\{w_2(v)|v \in V\}$ that *dominates* $\{w_1(v)|v \in V\}$—meaning that for every $v \in V$, $w_2(v) \geq w_1(v)$.

LEMMA 2.1. *In the memory allocation problem, let u_1 be a child of u_2 in the tree $G = (V, E)$. Let $\{w_1(v)|v \in V\}$ be an optimal memory basis. Assume the following condition is true: for every vertex $v \in Des(u_1)$, its* requirement set *$R(v) = \emptyset$; $R(u_1)$ contains an element (r, k), namely, $(r, k) \in R(u_1)$.*

Define S_1 as $S_1 = \mathcal{N}(u_2, r - d(u_2 \to u_1))$, and define S_2 as $S_2 = \mathcal{N}(u_1, r) - S_1$. We compute the elements of a set $\{w_2(v)|v \in V\}$ through the following three steps:

Step 1: *for all $v \in V$, let $w_2(v) \leftarrow w_1(v)$.*

Step 2: *Let*

$$X \leftarrow \max\left\{0, k - \sum_{v \in S_1} W_{\max}(v) - \sum_{v \in S_2} w_1(v)\right\},$$

and let $C \leftarrow S_2$.

Step 3: *Let v_0 be the vertex in C that is the closest to u_1—namely, $v_0 \in C$ and $d(v_0 \to u_1) = \min_{v \in C} d(v \to u_1)$. Let $w_2(v_0) \leftarrow \min\{W_{\max}(v_0), w_1(v_0) + X\}$. Let $X \leftarrow X - (w_2(v_0) - w_1(v_0))$, and let $C \leftarrow C - \{v_0\}$. Repeat Step* 3 *until X equals* 0*.*

Then the following two conclusions are true:

(1) *$\{w_2(v)|v \in V\}$ is an optimal memory basis, and $\{w_2(v)|v \in V\}$ dominates the optimal memory basis $\{w_1(v)|v \in V\}$;*

(2) *$\sum_{v \in S_1} W_{\max}(v) + \sum_{v \in S_2} w_2(v) \geq k$.*

PROOF. It is not difficult to see that the second conclusion is true. And it is simple to see that $\{w_2(v)|v \in V\}$ dominates the optimal memory basis $\{w_1(v)|v \in V\}$. So below we just need to prove that $\{w_2(v)|v \in V\}$ is an optimal memory basis.

The following two statements are clearly true:

STATEMENT 1: $S_1 \cup S_2 = \mathcal{N}(u_1, r)$, and $S_1 \cap S_2 = \emptyset$.

STATEMENT 2: For any $v \in S_2$, $W_{\min}(v) \leq w_1(v) \leq w_2(v) \leq W_{\max}(v)$. For any $v \in V - S_2$, $W_{\min}(v) \leq w_1(v) = w_2(v) \leq W_{\max}(v)$. And $\sum_{v \in V} w_2(v) - \sum_{v \in V} w_1(v) = \max\{0, k - \sum_{v \in S_1} W_{\max}(v) - \sum_{v \in S_2} w_1(v)\}$.

$\{w_1(v)|v \in V\}$ is an optimal memory basis. So there exists an optimal solution to the memory allocation problem that assigns memory size $w_{opt}(v)$ to every vertex $v \in V$, such that $w_1(v) \leq w_{opt}(v)$ for any $v \in V$. We know $\sum_{v \in \mathcal{N}(u_1, r)} w_{opt}(v) \geq k$. Since $\sum_{v \in \mathcal{N}(u_1, r)} w_{opt}(v) = \sum_{v \in S_1} w_{opt}(v) + \sum_{v \in S_2} w_{opt}(v) \leq \sum_{v \in S_1} W_{\max}(v) + \sum_{v \in S_2} w_{opt}(v)$, we get $\sum_{v \in S_2} w_{opt}(v) \geq k - \sum_{v \in S_1} W_{\max}(v)$. By *STATEMENT* 2, $\sum_{v \in S_2} w_2(v) = \sum_{v \in S_2} w_2(v) + \sum_{v \in V - S_2} w_2(v) - \sum_{v \in V - S_2} w_1(v) - \sum_{v \in S_2} w_1(v) + \sum_{v \in S_2} w_1(v) = \sum_{v \in V} w_2(v) - \sum_{v \in V} w_1(v) + \sum_{v \in S_2} w_1(v) = \max\{0, k - \sum_{v \in S_1} W_{\max}(v) - \sum_{v \in S_2} w_1(v)\} + \sum_{v \in S_2} w_1(v) = \max\{\sum_{v \in S_2} w_1(v), k - \sum_{v \in S_1} W_{\max}(v)\}$. So $\sum_{v \in S_2} w_{opt}(v) \geq \sum_{v \in S_2} w_2(v)$.

Let's compute a set $\{w_o(v)|v \in V\}$ through the following three steps:

Step 1: for all $v \in V - S_2$, let $w_o(v) \leftarrow w_{opt}(v)$. For all $v \in S_2$, let $w_o(v) \leftarrow w_1(v)$.

Step 2: Let $Y \leftarrow \sum_{v \in S_2} w_{opt}(v) - \sum_{v \in S_2} w_1(v)$, and let $C \leftarrow S_2$.

Step 3: Let v_0 be the vertex in C that is the closest to u_1—namely, $v_0 \in C$ and $d(v_0 \rightarrow u_1) = \min_{v \in C} d(v \rightarrow u_1)$. Let $w_o(v_0) \leftarrow \min\{W_{\max}(v_0), w_1(v_0) + Y\}$. Let $Y \leftarrow Y - (w_o(v_0) - w_1(v_0))$, and let $C \leftarrow C - \{v_0\}$. Repeat Step 3 until Y equals 0.

From the above three steps, it is simple to see that the following must be true: for any $v \in V$, $w_o(v) \geq w_2(v)$; for any $v \in V - S_2$, $w_o(v) = w_{opt}(v)$; for any $v \in S_2$, $W_{\min}(v) \leq w_o(v) \leq W_{\max}(v)$; $\sum_{v \in V} w_o(v) = \sum_{v \in V} w_{opt}(v)$, and $\sum_{v \in S_2} w_o(v) = \sum_{v \in S_2} w_{opt}(v)$. It is simple to see that the following must also be true: if there exists a vertex $v_1 \in S_2$ such that $w_o(v_1) > w_1(v_1)$, then for any $v \in S_2$ such that $d(v \rightarrow u_1) < d(v_1 \rightarrow u_1)$, $w_o(v) = W_{\max}(v)$; if there exists a vertex $v_2 \in S_2$ such that $w_o(v_2) < W_{\max}(v_2)$, then for any $v \in S_2$ such that $d(v \rightarrow u_1) > d(v_2 \rightarrow u_1)$, $w_o(v) = w_1(v)$. Therefore for any real number L, if we define Q as $Q = \{v|v \in S_2, d(v \rightarrow u_1) \leq L)\}$, then $\sum_{v \in Q} w_o(v) \geq \sum_{v \in Q} w_{opt}(v)$.

Let $v_0 \in V$ be any vertex such that $R(v_0) \neq \emptyset$, and let (r_0, k_0) be any element in $R(v_0)$. Clearly $v_0 \notin Des(u_1)$. Since $S_2 \subseteq Des(u_1) \cup \{u_1\}$, $\mathcal{N}(v_0, r_0) = \{v|v \in \mathcal{N}(v_0, r_0), v \notin S_2\} \cup \{v|v \in \mathcal{N}(v_0, r_0), v \in S_2\} = \{v|v \in \mathcal{N}(v_0, r_0), v \notin S_2\} \cup \{v|v \in S_2, d(v \rightarrow v_0) \leq r_0\} = \{v|v \in \mathcal{N}(v_0, r_0), v \notin S_2\} \cup \{v|v \in S_2, d(v \rightarrow u_1) \leq r_0 - d(u_1 \rightarrow v_0)\}$. So $\sum_{v \in \mathcal{N}(v_0, r_0)} w_o(v) \geq \sum_{v \in \mathcal{N}(v_0, r_0)} w_{opt}(v)$. Clearly $\sum_{v \in \mathcal{N}(v_0, r_0)} w_{opt}(v) \geq k_0$. So $\sum_{v \in \mathcal{N}(v_0, r_0)} w_o(v) \geq k_0$. Since $\sum_{v \in V} w_o(v) = \sum_{v \in V} w_{opt}(v)$, the memory-allocation solution that assigns memory size $w_o(v)$ to every vertex $v \in V$ is an optimal solution to the memory-allocation problem.

We have known that for any $v \in V$, $W_{\min}(v) \leq w_2(v) \leq w_o(v)$. So $\{w_2(v)|v \in V\}$ is an optimal memory basis. □

The following lemma shows how, given a memory-allocation problem, one can derive a new memory-allocation problem by modifying the *requirement sets* and *memory floors*, such that an optimal solution to the new problem is also an optimal solution to the original memory-allocation problem, and what's more, more vertices in the new problem have empty requirements sets than in the original problem (therefore the new problem is easier to solve).

LEMMA 2.2. *In the memory-allocation problem, let u_1 be a child of u_2 in the tree $G = (V, E)$. Let $\{w_0(v)|v \in V\}$ be an optimal memory basis. Assume the following conditions are true: for every vertex $v \in Des(u_1)$, its* requirement set *$R(v) = \emptyset$; for every element in $R(u_1)$—say the element is (r, k)—we have $\sum_{v \in \mathcal{N}(u_2, r - d(u_2 \rightarrow u_1))} W_{\max}(v) + \sum_{v \in \mathcal{N}(u_1, r) - \mathcal{N}(u_2, r - d(u_2 \rightarrow u_1))} w_0(v) \geq k$.*

We compute the elements of a set $\{\hat{R}(v)|v \in V\}$ through the following two steps:

Step 1*: for all $v \in V$, let $\hat{R}(v) \leftarrow R(v)$.*

Step 2*: let (r, k) be an element in $\hat{R}(u_1)$. If $\sum_{v \in \mathcal{N}(u_1, r)} w_0(v) < k$, then add an element $(r - d(u_2 \rightarrow u_1), k - \sum_{v \in \mathcal{N}(u_1, r) - \mathcal{N}(u_2, r - d(u_2 \rightarrow u_1))} w_0(v))$ to the set $\hat{R}(u_2)$. Remove the element (r, k) from $\hat{R}(u_1)$. Repeat Step 2 until $\hat{R}(u_1)$ becomes an empty set.*

Let's call the original memory-allocation problem, in which the requirement set *of each vertex $v \in V$ is $R(v)$, the* old problem. *We derive a new memory allocation problem—which we call the* new problem*—in the following way: in the new problem everything is the same as in the old problem, except that for each vertex $v \in V$, its* requirement set *is $\hat{R}(v)$ instead of $R(v)$, and its* memory floor *is $w_0(v)$ instead of $W_{\min}(v)$.*

Then the following two conclusions are true:

(1) *the new problem has a feasible solution;*

(2) *an optimal solution to the new problem is also an optimal solution to the old problem.*

Proof. It is not difficult to see that the first conclusion is true. Below we prove the second conclusion through two steps: first, we prove that an optimal solution to the new problem is a feasible solution to the old problem; then, we prove that an optimal solution to the new problem assigns the same *total memory size* to the vertices of the tree as an optimal solution to the old problem does.

Consider an optimal solution to the new problem that assigns *memory size* $\hat{w}_{opt}(v)$ to each vertex $v \in V$. Let $\bar{v} \in V$ be any vertex such that $R(\bar{v}) \neq \emptyset$, and let $(\bar{r}, \bar{k})$ be any element in $R(\bar{v})$. Either $(\bar{r}, \bar{k}) \in \hat{R}(\bar{v})$ or $(\bar{r}, \bar{k}) \notin \hat{R}(\bar{v})$. If $(\bar{r}, \bar{k}) \in \hat{R}(\bar{v})$, then clearly $\sum_{u \in \mathcal{N}(\bar{v}, \bar{r})} \hat{w}_{opt}(u) \geq \bar{k}$. Now consider the case where $(\bar{r}, \bar{k}) \notin \hat{R}(\bar{v})$. Clearly in this case $\bar{v} = u_1$, and either $\sum_{u \in \mathcal{N}(u_1, \bar{r})} w_0(u) \geq \bar{k}$, or $\sum_{u \in \mathcal{N}(u_1, \bar{r})} w_0(u) < \bar{k}$. If $\sum_{u \in \mathcal{N}(u_1, \bar{r})} w_0(u) \geq \bar{k}$, since $\hat{w}_{opt}(u) \geq w_0(u)$ for any $u \in V$, we have $\sum_{u \in \mathcal{N}(\bar{v}, \bar{r})} \hat{w}_{opt}(u) \geq \bar{k}$. We define S_1 as $S_1 = \mathcal{N}(u_2, \bar{r} - d(u_2 \to u_1))$, and define S_2 as $S_2 = \mathcal{N}(u_1, \bar{r}) - S_1$. Then if $\sum_{u \in \mathcal{N}(u_1, \bar{r})} w_0(u) < \bar{k}$, it is simple to see that $(\bar{r} - d(u_2 \to u_1), \bar{k} - \sum_{u \in S_2} w_0(u)) \in \hat{R}(u_2)$. So $\sum_{u \in \mathcal{N}(\bar{v}, \bar{r})} \hat{w}_{opt}(u) = \sum_{u \in S_1} \hat{w}_{opt}(u) + \sum_{u \in S_2} \hat{w}_{opt}(u) \geq \bar{k} - \sum_{u \in S_2} w_0(u) + \sum_{u \in S_2} \hat{w}_{opt}(u) \geq \bar{k}$. Therefore $\sum_{u \in \mathcal{N}(\bar{v}, \bar{r})} \hat{w}_{opt}(u) \geq \bar{k}$ in all cases. Therefore, an optimal solution to the new problem is a feasible solution to the old problem.

$\{w_0(v) | v \in V\}$ is an optimal memory basis for the old problem. So there exists an optimal solution to the old problem that assigns *memory size* $w_{opt}(v)$ to each vertex $v \in V$, such that for any $v \in V$, $w_0(v) \leq w_{opt}(v)$.

We compute the elements of four sets—$\{w_1(v) | v \in V\}$, $\{w_2(v) | v \in V\}$, $\{w_3(v) | v \in V\}$ and $\{w_4(v) | v \in V\}$—through the following five steps:

Step 1: for each $v \in Des(u_2)$, let $w_1(v) \leftarrow w_0(v)$. For each $v \in V - Des(u_2)$, let $w_1(v) \leftarrow w_{opt}(v)$.

Step 2: for each $v \in Des(u_2)$, let $w_2(v) \leftarrow w_{opt}(v) - w_0(v)$. For each $v \in V - Des(u_2)$, let $w_2(v) \leftarrow 0$.

Step 3: for each $v \in V$, let $w_3(v) \leftarrow 0$. Let $Z \leftarrow \sum_{v \in V} w_2(v)$, and let $C \leftarrow V$.

Step 4: Let v_0 be the vertex in C that is the closest to u_2—namely, $v_0 \in C$ and $d(v_0 \to u_2) = \min_{v \in C} d(v \to u_2)$. Let $w_3(v_0) \leftarrow \min\{W_{\max}(v_0) - w_1(v_0), Z\}$. Let $Z \leftarrow Z - w_3(v_0)$, and let $C \leftarrow C - \{v_0\}$. Repeat Step 4 until Z equals 0.

Step 5: for each $v \in V$, let $w_4(v) \leftarrow w_1(v) + w_3(v)$.

It is simple to see that the following must be true after the above five steps: $\sum_{v \in V} w_{opt}(v) = \sum_{v \in V} w_1(v) + \sum_{v \in V} w_2(v) = \sum_{v \in V} w_4(v)$, and $\sum_{v \in V} w_2(v) = \sum_{v \in V} w_3(v)$; for any $v \in V$, $w_0(v) \leq w_4(v) \leq W_{\max}(v)$; for any real number L, $\sum_{v\in\mathcal{N}(u_2,L)} w_3(v) \geq \sum_{v\in\mathcal{N}(u_2,L)} w_2(v)$; for any $v \in V$, if $w_4(v) < W_{\max}(v)$, then $\sum_{u\in\mathcal{N}(u_2,d(v\rightarrow u_2))} w_3(u) = \sum_{u\in V} w_2(u)$.

Let $\hat{v} \in V$ be any vertex such that$\hat{R}(\hat{v}) \neq \emptyset$, and let $(\hat{r}, \hat{k})$ be any element in$\hat{R}(\hat{v})$. Clearly $\hat{v} \in V - Des(u_2)$. Either $(\hat{r}, \hat{k}) \in R(\hat{v})$ or $(\hat{r}, \hat{k}) \notin R(\hat{v})$. If $(\hat{r}, \hat{k}) \in R(\hat{v})$, then $\sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_4(v) = \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_1(v) + \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_3(v) \geq \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_1(v) + \sum_{v\in\mathcal{N}(u_2,\hat{r}-d(u_2\rightarrow\hat{v}))} w_3(v) \geq \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_1(v) + \sum_{v\in\mathcal{N}(u_2,\hat{r}-d(u_2\rightarrow\hat{v}))} w_2(v) = \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_1(v) + \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_2(v) = \sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_{opt}(v) \geq \hat{k}$.

Now consider the case where $(\hat{r}, \hat{k}) \notin R(\hat{v})$. We define $\tilde{r}$ as $\tilde{r} = \hat{r} + d(u_2 \rightarrow u_1)$, define $\hat{S}_1$ as $\hat{S}_1 = \mathcal{N}(u_2, \hat{r})$, define $\hat{S}_2$ as $\hat{S}_2 = \mathcal{N}(u_1, \tilde{r}) - \hat{S}_1$, and define $\tilde{k}$ as $\tilde{k} = \hat{k} + \sum_{v\in\hat{S}_2} w_0(v)$. It is easy to see that, in this case, $\hat{v} = u_2$ and $(\tilde{r}, \tilde{k}) \in R(u_1)$. If $w_4(v) = W_{\max}(v)$ for every vertex $v \in \mathcal{N}(u_2, \hat{r})$, then clearly $\sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_4(v) \geq \hat{k}$ because the new problem has a feasible solution. If there exists $v_0 \in \mathcal{N}(u_2, \hat{r})$ such that $w_4(v_0) < W_{\max}(v_0)$, then $\sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_4(v) = \sum_{v\in\hat{S}_1} w_1(v) + \sum_{v\in V} w_2(v) \geq \sum_{v\in\mathcal{N}(u_1,\tilde{r})} w_1(v) - \sum_{v\in\hat{S}_2} w_1(v) + \sum_{v\in\mathcal{N}(u_1,\tilde{r})} w_2(v) = \sum_{v\in\mathcal{N}(u_1,\tilde{r})} w_{opt}(v) - \sum_{v\in\hat{S}_2} w_1(v) \geq \tilde{k} - \sum_{v\in\hat{S}_2} w_1(v) = \hat{k}$.

So $\sum_{v\in\mathcal{N}(\hat{v},\hat{r})} w_4(v) \geq \hat{k}$ in all cases. So the solution that assigns *memory size* $w_4(v)$ to every vertex $v \in V$ is a feasible solution to the new problem—so $\sum_{v\in V} \hat{w}_{opt}(v) \leq \sum_{v\in V} w_4(v)$. Since every optimal solution to the new problem is a feasible solution to the old problem, we have $\sum_{v\in V} w_{opt}(v) \leq \sum_{v\in V} \hat{w}_{opt}(v)$. Clearly $\sum_{v\in V} w_4(v) = \sum_{v\in V} w_{opt}(v)$, so $\sum_{v\in V} \hat{w}_{opt}(v) = \sum_{v\in V} w_{opt}(v)$. So the optimal solution to the new problem, which assigns memory size $\hat{w}_{opt}(v)$ to every vertex $v \in V$, is also an optimal solution to the old problem. Now we can see that the second conclusion of this lemma is true. □

Lemma 2.1 and Lemma 2.2 naturally lead us to an algorithm for optimally solving the memory-allocation problem. We can process the vertices of the tree one by one, with every vertex processed before its parent. Every time a vertex is processed, corresponding to each element in its requirement set, we use the method in Lemma 2.1 to derive an optimal memory basis of larger values; then we use the method in Lemma 2.2 to force the vertex's requirement set to be empty. In the end, the root becomes the only vertex whose requirement set may not be empty, and the memory-allocation problem becomes very simple to solve.

The following algorithm outputs an *optimal* solution to the memory-allocation problem.

Algorithm 2.1 (Memory-Allocation on Tree $G = (V, E)$)

(1) Initially, for every vertex $v \in V$, let $w(v) \leftarrow W_{\min}(v)$.

(2) Process all the vertices one by one, in an order that follows the following rule: every vertex is processed before its parent. For each vertex $\tilde{v} \in V$ that is not the root v_{root}, it is processed through the following two steps:

Step 1: Treat $\tilde{v}$, the parent of $\tilde{v}$ and the set $\{w(v)|v \in V\}$, respectively, as the vertex u_1, the vertex u_2 and the set $\{w_1(v)|v \in V\}$ in Lemma 2.1, and for each element in $R(\tilde{v})$ do the following two things: (1) treat that element in $R(\tilde{v})$ as the element (r, k) in Lemma 2.1, and compute the set $\{w_2(v)|v \in V\}$ as in Lemma 2.1; (2) for every vertex $v \in V$, let $w(v)$ get the value of $w_2(v)$—namely, $w(v) \leftarrow w_2(v)$.

Step 2: Treat $\tilde{v}$, the parent of $\tilde{v}$ and the set $\{w(v)|v \in V\}$, respectively, as the vertex u_1, the vertex u_2 and the set $\{w_0(v)|v \in V\}$ in Lemma 2.2, and do the following two things: (1) compute the set $\{\hat{R}(v)|v \in V\}$ as in Lemma 2.2; (2) for every vertex $v \in V$, let $R(v) \leftarrow \hat{R}(v)$, and let $W_{\min}(v) \leftarrow w(v)$.

The root v_{root} is processed in the following way:

Pretend that the root v_{root} has a parent whose distance to v_{root} is infinitely large. Treat v_{root} as the vertex $\tilde{v}$ above, and run just its Step 1.

(3) Output the following solution as the solution to the memory-allocation problem: for each vertex $v \in V$, assign $w(v)$ to it as its "memory size."

A pseudocode of Algorithm 2.1 is presented in Appendix A for interested readers.

Analysis shows that Algorithm 2.1 has time complexity $O(q|V|^3)$, where $|V|$ is the number of vertices and q is the average cardinality of a requirement set, namely, $q = \frac{1}{|V|}\sum_{v \in V} |R(v)|$. The complexity analysis as well as the proof for the correctness of Algorithm 2.1 (including the optimality of the solution it outputs) are presented in Appendix B.

2.3 Variation of the Algorithm

Algorithm 2.1 has complexity $O(q|V|^3)$. But if $W_{\max}(v) = \infty$ for all $v \in V$—namely, if no upper bound exists for the vertices' memory sizes—then an algorithm of time complexity $O(q|V|^2)$ can actually be derived. We present such an algorithm and its complexity analysis in Appendix C for interested readers.

3. DATA INTERLEAVING

3.1 Definition of the Problem

Assume that, in the file storage problem, the number of codeword symbols assigned to each vertex $v \in V$, $w(v)$, is already known. (The only requirement for $w(v)$ here is that for every vertex $u \in V$ and for $1 \le i \le n_u$, $\sum_{v \in \mathcal{N}(u, r_i(u))} w(v) \ge k_i(u)$. No feasible solution to the problem exists if that requirement is not satisfied.) Then, the file storage problem is simplified to be the following data-interleaving problem.

Definition 3.1. *The Date-Interleaving Problem.*

Instance: A tree $G = (V, E)$ with asymmetric edges, and N different colors. Every edge $(u, v) \in E$ has a positive length $l(u, v)$. Every vertex $v \in V$ is associated with a set $R(v) = \{(r_i(v), k_i(v)) | 1 \le i \le n_v\}$, which is called the *requirement set of* v. Every vertex $v \in V$ is also associated with a nonnegative integer $w(v)$. (Here $w(v) \le N$.)

Question: How to assign $w(v)$ colors to each vertex $v \in V$, such that for every vertex $u \in V$ and for $1 \le i \le n_u$, the vertices in the set $\mathcal{N}(u, r_i(u))$ together have at least $k_i(u)$ distinct colors? (At most N different colors can be used, and every color can be assigned more than once to the vertices.)

Comments: All the parameters above have the same meaning as in the file storage problem (Definition 1.1). So we omit defining their allowed ranges of values.

In the data-interleaving problem, we use N different colors to represent the N symbols in the codeword, for a more abstract understanding of the problem.

3.2 Data-Interleaving Algorithm

In the Jiang and Bruck [2003a], a solution was presented for coloring the vertices of an undirected tree using N colors, in such a way that, for every point of the tree (which can be either a vertex or a point on an edge), there exist K different colors that are placed as closely as possible around it, for a preset parameter K ($K \leq N$). In this section, we derive a data-interleaving algorithm using a similar technique. The new algorithm is adapted to the file-allocation scheme studied here, and is for trees with asymmetric edges between adjacent vertices.

Since every vertex $v \in V$ is to be assigned $w(v)$ colors, we think of v as having $w(v)$ *color-slots*, where each color-slot is to be assigned one color—and we say that those $w(v)$ color-slots *belong to* v.

We define ϖ as $\varpi = \sum_{v \in V} w(v)$, that is, the total number of color-slots in the tree G. We label all the color-slots in G as $s_1, s_2, \ldots, s_\varpi$ following this rule: if $d(s_i \rightarrow v_{root}) < d(s_j \rightarrow v_{root})$, then $i < j$.

For any two color-slots s_i and s_j, we use $d(s_i \rightarrow s_j)$ to denote the distance from the vertex that s_i belongs to, to the vertex that s_j belongs to. In other words, if s_i is a color-slot of vertex u, and s_j is a color-slot of vertex v, then $d(s_i \rightarrow s_j) = d(u \rightarrow v)$. Similarly, we also use $d(s_i \rightarrow v)$ and $d(u \rightarrow s_j)$ to denote the same value as $d(u \rightarrow v)$.

For any vertex $v \in V$ and any real number r, we define $B(v,r)$ as $B(v,r) = \{s_i | 1 \leq i \leq \varpi, d(s_i \rightarrow v) \leq r\}$—namely, the set of color-slots whose distance to v is at most r. Similarly, for any color-slot s_j and any real number r, we define $B(s_j, r)$ as $B(s_j, r) = \{s_i | 1 \leq i \leq \varpi, d(s_i \rightarrow s_j) \leq r\}$.

For any three color-slots s_x, s_y, and s_z, where $x \neq y$ (but z does not have to be different from x and y), we use "$(s_x \Rightarrow s_z) \lhd (s_y \Rightarrow s_z)$" to denote the following condition: either $d(s_x \rightarrow s_z) < d(s_y \rightarrow s_z)$, or $d(s_x \rightarrow s_z) = d(s_y \rightarrow s_z)$ and $x < y$.

Similarly, by replacing the term *color-slot* s_z in the above paragraph by *vertex* v, we get the definition of $(s_x \Rightarrow v) \lhd (s_y \Rightarrow v)$.

For every vertex v, we define κ_v as $\kappa_v = \max_{1 \leq i \leq n_v} k_i(v)$. We use S_v to denote the set of color-slots that satisfies the following two conditions: (1) $|S_v| = \kappa_v$; (2) for any color-slot $s_p \in S_v$ and any color-slot $s_q \notin S_v$, $(s_p \Rightarrow v) \lhd (s_q \Rightarrow v)$.

Finally, for every color-slot s_i, we assign to it an integer X_i that satisfies the following two conditions: (1) for every vertex v, if $s_i \in S_v$, then $X_i \geq |S_v \cap \{s_t | t \leq i\}|$; (2) $1 \leq X_i \leq N$, and $X_i \leq i$. For now let's assume that the integer X_i is given by an oracle; later in Subsection 3.3 we will discuss how to set the value of X_i.

The following algorithm solves the data interleaving problem.

```
Algorithm 3.1 (Data Interleaving on Tree $G = (V, E)$)
for $i = 1$ to $\varpi$ do
{Let $T$ be the set of color-slots that satisfies the following two conditions:
      (1) $T \subseteq \{s_t | t < i\}$, and $|T| = X_i - 1$;
      (2) for any color-slot $s_p \in T$ and any color-slot $s_q \in \{s_t | t < i\} - T$,
           $(s_p \Rightarrow s_i) \lhd (s_q \Rightarrow s_i)$.
   Assign to $s_i$ a color that differs from the color of every color-slot in $T$.
}
```

LEMMA 3.1. *After Algorithm* 3.1 *is used to assign colors to the tree* $G = (V, E)$*, for any integer* i $(1 \leq i \leq \varpi)$ *and any vertex* $v \in V$*, no two color-slots in the set* $S_v \cap \{s_t | t \leq i\}$ *are assigned the same color.*

PROOF. Let v be an arbitrary vertex. Let's use *ASSERTION* to denote the following assertion: no two color-slots in the set $S_v \cap \{s_t | t \leq i\}$ are assigned the same color.

We use induction on the parameter i ($1 \leq i \leq \varpi$) to prove this lemma.

When $i = 1$, the set $S_v \cap \{s_t | t \leq i\}$ contains at most one color-slot, so the *ASSERTION* is true. This serves as our base case.

Now let I be an integer such that $2 \leq I \leq \varpi$. Assume that when $i < I$, the *ASSERTION* is true. We shall prove that when $i = I$, the *ASSERTION* still holds.

Let $i = I$. If $|S_v \cap \{s_t | t \leq i\}|$ equals 0 or 1, then clearly the *ASSERTION* is true. If $|S_v \cap \{s_t | t \leq i\}| \geq 1$ and $s_i \notin S_v \cap \{s_t | t \leq i\}$, by letting j^* denote the maximum value of j subject to the constraint that $s_j \in S_v \cap \{s_t | t \leq i\}$, we can see that $S_v \cap \{s_t | t \leq i\} = S_v \cap \{s_t | t \leq j^*\}$ and $j^* < i$—then by the induction assumption, no two color-slots in the set $S_v \cap \{s_t | t \leq j^*\}$ are assigned the same color, so the *ASSERTION* is true. Therefore in the remainder of the proof, we shall only consider the case where $|S_v \cap \{s_t | t \leq i\}| \geq 2$ and $s_i \in S_v \cap \{s_t | t \leq i\}$.

Let *LCA* denote the *least common ancestor* of v and the vertex that s_i belongs to—namely, *LCA* is the unique vertex that lies on the path between v_{root} and v, on the path between v_{root} and the vertex that s_i belongs to, and on the path between v and the vertex that s_i belongs to.

Let s_p be an arbitrary color-slot in the set $S_v \cap \{s_t | t < i\}$. Let s_q be an arbitrary color-slot in the set $\{s_t | s_t \notin S_v, t < i\}$. Define P as such a set: $P = \{s_t | t < i,$ *the vertex that* s_t *belongs to is either LCA or a descendant of LCA*$\}$. We have the following three statements.

STATEMENT 1: $s_q \notin P$.

STATEMENT 2: if $s_p \in P$, then $(s_p \Rightarrow s_i) \lhd (s_q \Rightarrow s_i)$.

STATEMENT 3: if $s_p \notin P$, then $(s_p \Rightarrow s_i) \lhd (s_q \Rightarrow s_i)$.

To see why *STATEMENT* 1 is true, we use contradiction. Assume $s_q \in P$. Then $d(s_q \rightarrow v_{root}) = d(s_q \rightarrow LCA) + d(LCA \rightarrow v_{root})$. Clearly $d(s_i \rightarrow v_{root}) = d(s_i \rightarrow LCA) + d(LCA \rightarrow v_{root})$. Since $q < i$, we get that $d(s_q \rightarrow v_{root}) \leq d(s_i \rightarrow v_{root})$, so $d(s_q \rightarrow LCA) \leq d(s_i \rightarrow LCA)$. So $d(s_q \rightarrow v) \leq d(s_q \rightarrow LCA) + d(LCA \rightarrow v) \leq d(s_i \rightarrow LCA) + d(LCA \rightarrow v) = d(s_i \rightarrow v)$. Since $s_i \in S_v$, $d(s_q \rightarrow v) \leq d(s_i \rightarrow v)$ and $q < i$, by the definition of S_v, we get that $s_q \in S_v$, which is a contradiction. So *STATEMENT* 1 is true.

To see why *STATEMENT* 2 is true, let's assume that $s_p \in P$. By the same argument as in the previous paragraph, we get that $d(s_p \rightarrow LCA) \leq d(s_i \rightarrow LCA)$. Since $s_i \in S_v$, $i > q$ and $s_q \notin S_v$, by the definition of S_v, we get that $d(s_q \rightarrow v) > d(s_i \rightarrow v)$. From *STATEMENT* 1, we know that the vertex that s_q belongs to is neither *LCA* nor a descendant of *LCA*, so $d(s_q \rightarrow v) = d(s_q \rightarrow LCA) + d(LCA \rightarrow v)$; and we know that $d(s_i \rightarrow v) = d(s_i \rightarrow LCA) + d(LCA \rightarrow v)$. So $d(s_q \rightarrow LCA) > d(s_i \rightarrow LCA) \geq d(s_p \rightarrow LCA)$. So $d(s_p \rightarrow s_i) \leq d(s_p \rightarrow LCA) + d(LCA \rightarrow s_i) < d(s_q \rightarrow LCA) + d(LCA \rightarrow s_i) = d(s_q \rightarrow s_i)$. So $(s_p \Rightarrow s_i) \lhd (s_q \Rightarrow s_i)$. So *STATEMENT* 2 is true.

To see why *STATEMENT* 3 is true, let's assume that $s_p \notin P$. Since $s_p \in S_v$ and $s_q \notin S_v$, by the definition of S_v, we get that $(s_p \Rightarrow v) \lhd (s_q \Rightarrow v)$. Since neither the vertex that s_p belongs to nor the vertex that s_q belongs to is *LCA* or a descendant of *LCA*, but v is either the same as or a descendant of *LCA*, we get that $(s_p \Rightarrow LCA) \lhd (s_q \Rightarrow LCA)$. Since s_i is either the same as or a descendant of *LCA*, we get that $(s_p \Rightarrow s_i) \lhd (s_q \Rightarrow s_i)$. So *STATEMENT* 3 is true.

By *STATEMENT* 2 and *STATEMENT* 3, we know that $(s_p \Rightarrow s_i) \lhd (s_q \Rightarrow s_i)$ in any case. Note that s_p is an arbitrary color-slot in the set $S_v \cap \{s_t | t < i\}$, and s_q is an arbitrary color-slot in the set $\{s_t | s_t \notin S_v, t < i\}$. Since $X_i \geq |S_v \cap \{s_t | t \leq i\}|$ and $s_i \in S_v$, we get that $|S_v \cap \{s_t | t < i\}| \leq X_i - 1$—so by Algorithm 3.1, the color of s_i differs from that of any color-slot in $S_v \cap \{s_t | t < i\}$. By the induction assumption, no two color-slots in the set $S_v \cap \{s_t | t < i\}$ are assigned the same color. So no two color-slots in $S_v \cap \{s_t | t \leq i\}$ are assigned the same color. So the *ASSERTION* is true when $i = I$. That concludes the induction step of the proof. □

Lemma 3.2. *After Algorithm* 3.1 *is used to assign colors to the tree* $G = (V, E)$*, for any vertex* v*, no two color-slots in* S_v*—whose cardinality is* $\kappa_v = \max_{1 \leq i \leq n_v} k_i(v)$*—are assigned the same color.*

Proof. Replace the integer i in Lemma 3.1 by ϖ. Note that $S_v \cap \{s_t | t \leq \varpi\} = S_v$. □

Theorem 3.3. *Algorithm* 3.1 *correctly outputs a solution to the data interleaving problem.*

Proof. For any vertex v, we know that there are $\kappa_v = \max_{1 \leq i \leq n_v} k_i(v)$ distinct colors assigned to the color-slots in S_v. Consider an arbitrary *requirement* of v—say it is $(r_i(v), k_i(v))$. By the definition of S_v and the fact that $|B(v, r_i(v))| \geq k_i(v)$, we can see that there are at least k_i distinct colors assigned to the color-slots in the set $\{s_t | d(s_t \to v) \leq r_i\}$. □

3.3 Discussion of the Data-Interleaving Algorithm

For Algorithm 3.1, the minimum value that X_i $(1 \leq i \leq \varpi)$ can take is the greater number between 1 and $\max_{v: s_i \in S_v} |S_v \cap \{s_t | t \leq i\}|$, and the maximum value X_i can take is $\min\{N, i\}$. X_i can take any value between those two bounds. The smaller X_i is, the less restriction the algorithm has while choosing a color for s_i—therefore the more possible outputs the algorithm has. So choosing a smaller value for X_i increases the generality of the algorithm; on the other side, setting X_i to be the maximum value—$\min\{N, i\}$—certainly makes its computation simple.

If we set X_i to be $\min\{N, i\}$ for all i, then Algorithm 3.1 will output a solution that has the following property: for every vertex, there are N different colors placed as closely to it as possible. That property can be proved by using the following two facts: (1) if we make the requirement set of each vertex $v \in V$ to be $R(v) = \{(*, N)\}$, where $*$ is an arbitrary integer, Algorithm 3.1 will still work exactly the same way as before (since the value of each X_i has been fixed to be $\min\{N, i\}$); (2) Lemma 3.2 tells us that, if a vertex has a requirement (r, k), then Algorithm 3.1 places at least k different colors as closely to it as possible.

Note that, for the data-interleaving algorithm, we can, in fact, pick any vertex of G to be the root vertex v_{root}. It does not have to be the same root as in the memory-allocation algorithm.

The complexity analysis of Algorithm 3.1 is presented in Appendix D.

4. OPTIMAL SOLUTION TO THE FILE STORAGE PROBLEM

The combination of the memory-allocation algorithm and the data-interleaving algorithm yields an optimal solution to the file storage problem—we first use the memory-allocation algorithm to determine the number of codeword symbols, $w(v)$, assigned to each vertex $v \in V$ (where we should set $W_{\min}(v) = 0$ for all $v \in V$ in the corresponding memory-allocation problem), then use the data-interleaving algorithm to determine which $w(v)$ codeword symbols to assign to each vertex $v \in V$.

For any element (r, k) in the requirement set of any vertex $v \in V$, the memory-allocation algorithm guarantees that there are at least k codeword symbols placed within distance r to v; then the data-interleaving algorithm further guarantees that there are at least *k distinct* codeword symbols placed within distance r to v—so the solution to the file storage problem is feasible. The total memory size determined by the memory-allocation algorithm, $\sum_{v\in V} w(v)$, is a lower bound for the total memory size in a file storage solution, and the data-interleaving algorithm shows that this lower bound is in fact sufficiently large—so the solution to the file storage problem is optimal.

5. CONCLUSION

This article proposes a scheme for storing a file in a network where clients have diverse requirements on file-retrieval delays, under both fault-free and faulty circumstances. The file is encoded with a general error-correcting code. When the network is a tree with asymmetric edges between adjacent nodes, a memory-allocation algorithm and a data-interleaving algorithm are used to respectively determine how many and which codeword symbol to store on each node. Both algorithms are of polynomial time complexity. They together provide an optimal solution to the file storage problem, which minimizes the total amount of data stored in the network.

There are many additional important issues to be solved in the field of file storage using error-correcting codes. Among them, storing files in dynamic environments and finding good codes that have low complexity for file revision are two interesting examples.

APPENDIX A. PSEUDOCODE OF ALGORITHM 2.1

In this appendix we present the pseudocode of Algorithm 2.1.

Algorithm 2.1 (Memory Allocation on Tree $G = (V, E)$)

```
1. Label the vertices in $V$ as $v_1, v_2, \ldots, v_{|V|}$ according to the following rule: if $v_i$
   is the parent of $v_j$, then $i > j$.
   For $1 \le i \le |V|$, let $w(v_i) \leftarrow W_{\min}(v_i)$.
2. For $i = 1$ to $|V| - 1$ do:
   { Let $v_P$ denote the parent of $v_i$. Let $\tilde{R}(v_i) \leftarrow R(v_i)$.
     While $\tilde{R}(v_i) \neq \emptyset$ do:
     { Let $(r, k)$ be any element in $\tilde{R}(v_i)$. Define $S_1$ as $S_1 = \mathcal{N}(v_P, r - d(v_P \to v_i))$,
       and define $S_2$ as $S_2 = \mathcal{N}(v_i, r) - S_1$. Update the elements in $\{w(v) | v \in V\}$
       through the following two steps:
           Step 1: Let $X \leftarrow \max\{0, k - \sum_{v\in S_1} W_{\max}(v) - \sum_{v\in S_2} w(v)\}$, and let
                   $C \leftarrow S_2$.
```

```
            Step 2: Let $u_0$ be the vertex in $C$ that is the closest to $v_i$—namely, $u_0 \in C$
                    and $d(u_0 \to v_i) = \min_{u \in C} d(u \to v_i)$. Let $Temp \leftarrow \min\{W_{\max}(u_0),$
                    $w(u_0) + X\}$. Let $X \leftarrow X - (Temp - w(u_0))$, let $w(u_0) \leftarrow Temp$,
                    and let $C \leftarrow C - \{u_0\}$. Repeat Step 2 until $X$ equals 0.
          Remove the element $(r, k)$ from $\tilde{R}(v_i)$.
        }
        While $R(v_i) \neq \emptyset$ do:
        { Let $(r, k)$ be any element in $R(v_i)$. If $\sum_{u \in \mathcal{N}(v_i, r)} w(u) < k$, then add an
          element $(r - d(v_P \to v_i), k - \sum_{u \in \mathcal{N}(v_i, r) - \mathcal{N}(v_P, r - d(v_P \to v_i))} w(u))$ to the set
          $R(v_P)$. Remove the element $(r, k)$ from $R(v_i)$.
        }
      }
  3. While $R(v_{|V|}) \neq \emptyset$ do:
     { Let $(r, k)$ be any element in $R(v_{|V|})$. Update the elements in $\{w(v) | v \in V\}$
       through the following two steps:
         Step 1: Let $X \leftarrow \max\{0, k - \sum_{u \in \mathcal{N}(v_{|V|}, r)} w(v)\}$, and let $C \leftarrow V$.
         Step 2: Let $u_0$ be the vertex in $C$ that is the closest to $v_{|V|}$—namely,
                 $u_0 \in C$ and $d(u_0 \to v_{|V|}) = \min_{u \in C} d(u \to v_{|V|})$. Let $Temp \leftarrow$
                 $\min\{W_{\max}(u_0), w(u_0) + X\}$. Let $X \leftarrow X - (Temp - w(u_0))$, let
                 $w(u_0) \leftarrow Temp$, and let $C \leftarrow C - \{u_0\}$. Repeat Step 2 until $X$
                 equals 0.
       Remove the element $(r, k)$ from $R(v_{|V|})$.
     }
     Output $w(v_1), w(v_2), \ldots, w(v_{|V|})$ as the solution to the memory-allocation
      problem.
```

Note that in the above pseudocode, the values of *memory floors* are not really updated because it is not necessary to do that, although they have been used in Section 2 as a helpful tool for analysis.

APPENDIX B. PROOF AND COMPLEXITY ANALYSIS OF ALGORITHM 2.1

In this appendix, we prove the correctness of Algorithm 2.1, and analyze its complexity.

THEOREM B.1. *Algorithm* 2.1 *correctly outputs an optimal solution to the memory allocation problem.*

PROOF. For all the vertices except the root v_{root}, Algorithm 2.1 processes them one by one, using the methods in Lemma 2.1 and Lemma 2.2 to increase the memory sizes of vertices and transform the memory allocation problem from old problems to new problems. (To recall the definition of *old problem* and *new problem*, see Lemma 2.2.) After that, only v_{root} has not been processed, and v_{root} is the only vertex whose requirement set may not be empty. Then the algorithm increases the memory sizes of the vertices to satisfy v_{root}'s requirements, with the increased part of the memory sizes placed as close as possible to v_{root} and being as small as possible, and ends there—and that is clearly the optimal way to solve the new memory-allocation problem at that moment (which is just to assign enough memory sizes to satisfy the requirements of v_{root}). Since an optimal solution to a new problem is always an optimal solution to an old problem, Algorithm 2.1 has successfully found an optimal solution to the original memory-allocation problem. □

In terms of complexity analysis, Algorithm 2.1 needs two tools for its execution: a distance matrix recording the distance between any pair of vertices, which takes time complexity $O(|V|^2)$ to compute; and for every vertex v, a table ordering all the vertices according to their distance to v—computing all these $|V|$ tables has time complexity $O(|V|^2)$, too. With these two tools available, the algorithm processes all the vertices one by one. Let q denote the average cardinality of a requirement set in the original memory-allocation problem, namely, $q = \frac{1}{|V|}\sum_{v\in V}|R(v)|$. So originally there are totally $q|V|$ elements in all the requirement sets. When the algorithm is computing, every time an element in a vertex's requirement set is deleted, a new element might be inserted into the vertex's parent's requirement set—and on no other occasion will a new element be generated. Each vertex can have at most $|V| - 1$ ancestors. So during the whole period when the algorithm is computing, there are no more than $q|V|^2$ elements—old and new, in total—in all the requirement sets. Every time a vertex is processed, all the elements in its requirement sets are processed in the following way—for each element, the set $\{w(v)|v \in V\}$ and the set $\{R(v)|v \in V\}$ are updated, which has time complexity $O(|V|)$. So the complexity of Algorithm 2.1 is $O(|V|^2 + |V|^2 + q|V|^2 \cdot |V|)$, which equals $O(q|V|^3)$.

APPENDIX C. ALGORITHM FOR MEMORY-ALLOCATION PROBLEM WITHOUT UPPER BOUND FOR MEMORY SIZES

When $W_{\max}(v) = \infty$ for all $v \in V$—that is, when no upper bound exists for the memory sizes—the memory-allocation problem can be solve with time complexity $O(q|V|^2)$. In this appendix, we present the pseudocode of such an algorithm—Algorithm C.1.

Algorithm C.1 is similar to Algorithm 2.1, except that in Algorithm C.1, a new notion named *residual requirement set* is used. The notion is defined as follows. Say at some moment, each vertex $v \in V$ is temporarily assigned a memory size $w(v)$, and its *requirement set* is $R(v)$. For every element $(r, k) \in R(v)$, there is a corresponding element $(\bar{r}, \bar{k})$ in the *residual requirement set of* v, denoted by $Res(v)$, computed in the following way: $\bar{r} = r$, and $\bar{k} = \max\{k - \sum_{u\in\mathcal{N}(v,r)} w(u), 0\}$. (The meaning of the element $(\bar{r}, \bar{k})$ is that the summation of the memory sizes of the vertices in $\mathcal{N}(v, r)$ needs to be increased by $\bar{k}$ so that $\sum_{u\in\mathcal{N}(v,r)} w(u)$ will be no less than k.)

Algorithm C.1 (Memory-Allocation on Tree $G = (V, E)$ without Upper Bound for Memory Sizes)

```
1. Label the vertices in $V$ as $v_1, v_2, \dots, v_{|V|}$ according to the following rule: if
   $v_i$ is the parent of $v_j$, then $i > j$. Let $w(v_i) \leftarrow W_{\min}(v_i)$ for $1 \le i \le |V|$. Let
   $Res(v_i) \leftarrow \emptyset$ for $1 \le i \le |V|$. For $1 \le i \le |V|$, and for each element $(r, k) \in R(v_i)$,
   do the following: if $k - \sum_{v\in\mathcal{N}(v_i,r)} w(v) > 0$, then let $Res(v_i) \leftarrow Res(v_i) \cup$
   $\{(r, k - \sum_{v\in\mathcal{N}(v_i,r)} w(v))\}$.
2. For $i = 1$ to $|V| - 1$ do:
   { Let $v_P$ denote the parent of $v_i$. Let $Q(v_i) \leftarrow Res(v_i)$, and let $x \leftarrow 0$.
     While $Q(v_i) \ne \emptyset$ do:
     { Let $(r, k)$ be any element in $Q(v_i)$. If $r < d(v_P \rightarrow v_i)$, then let $x \leftarrow \max\{x, k\}$
       and remove the element $(r, k)$ from the set $Res(v_i)$. Remove the element
       $(r, k)$ from $Q(v_i)$.
     }
```

```
    Let $w(v_i) \leftarrow w(v_i) + x$.
    For $j = i + 1$ to $|V|$, and for every element $(r, k) \in Res(v_j)$, do the following:
     if $r \geq d(v_i \to v_j)$ and $k - x > 0$, then replace the element $(r, k)$ in the set
    $Res(v_j)$ by $(r, k - x)$; if $r \geq d(v_i \to v_j)$ and $k - x \leq 0$, then remove the element
    $(r, k)$ from $Res(v_j)$.
    For every element $(r, k) \in Res(v_i)$ do the following: if $k > x$, then let $Res(v_P)$
    $\leftarrow Res(v_P) \cup \{(r - d(v_P \to v_i), k - x)\}$.
    Let $Res(v_i) \leftarrow \emptyset$.
  }
3. Let $x \leftarrow 0$.
   While $Res(v_{|V|}) \neq \emptyset$ do:
   { Let $(r, k)$ be any element in $Res(v_{|V|})$. Let $x \leftarrow \max\{x, k\}$. Remove the
     element $(r, k)$ from $Res(v_{|V|})$.
   }
   Let $w(v_{|V|}) \leftarrow w(v_{|V|}) + x$.
4. Output $w(v_1)$, $w(v_2)$, ..., $w(v_{|V|})$ as the solution to the memory allocation problem.
```

In regard to the complexity analysis, the complexity of Algorithm 2.1, which is $O(q|V|^3)$, is dominated by the complexity of updating memory sizes—the memory sizes can be updated up to $O(q|V|^2)$ times, and each time up to $O(|V|)$ memory sizes might change. When there is no upper bound for the memory sizes, with the help of residual requirement sets, each time only one memory size will need to be updated, which has complexity $O(1)$. So the complexity of updating memory sizes is reduced from $O(q|V|^3)$ to $O(q|V|^2)$. Maintaining the residual requirement sets also has a total complexity of $O(q|V|^2)$. So the complexity of Algorithm C.1 is $O(q|V|^2)$.

APPENDIX D. COMPLEXITY OF THE DATA-INTERLEAVING ALGORITHM

The complexity of the data-interleaving algorithm depends on how the variables X_i $(1 \leq i \leq \varpi)$ are chosen. The smaller the values of X_i are, the more general the algorithm is—meaning that the algorithm has more possible outputs. The smallest value X_i can take is $\max_{v:s_i \in S_v} |S_v \cap \{s_t | t \leq i\}|$ (assuming that number is no less than 1; otherwise the value is simply 1.) Below we will show that, if the algorithm chooses X_i to be $\max_{v:s_i \in S_v} |S_v \cap \{s_t | t \leq i\}|$ for $1 \leq i \leq \varpi$, then the algorithm has the total time complexity of $O(|V|^2 + N\varpi^2 + N\varpi|V|)$. We point out that this time complexity can be reduced if one is willing to add more restrictions on the algorithm—for example, when the color-slots of the same vertex are labeled with consecutive indices, or when X_i is simply set to be $\min\{N, i\}$ for all i, the algorithm can be implemented in more efficient ways.

The full implementation of the data-interleaving algorithm has the following major operations:

—*Operation* 1: label the color-slots as $s_1, s_2, \ldots, s_\varpi$;

—*Operation* 2: for each vertex v, find out the set S_v;

—*Operation* 3: for $1 \leq i \leq \varpi$, set the value of X_i to be $\max_{v:s_i \in S_v} |S_v \cap \{s_t | t \leq i\}|$;

—*Operation* 4: for $1 \leq i \leq \varpi$, find the set denoted by T in the algorithm.

Below we analyze the time complexity.

In order to implement the algorithm, we need to construct a $|V| \times |V|$ distance table which records the distance from any vertex to any other vertex; then for every vertex v, we need to construct a list which orders all the vertices according to their distance to v. That has time complexity $O(|V|^2)$. Then, *Operation* 1 has time complexity $O(\varpi + |V|)$.

Now consider *Operation* 2. Computing $\kappa_v = \max_{1\leq i\leq n_v} k_i(v)$ for all v takes complexity $\sum_{v\in V} n_v$. It is very reasonable to assume that $n_v \leq N$ (because otherwise some of v's *requirements* would be redundant), so that complexity becomes $O(N|V|)$. To find out S_v, we need to sort the color-slots first based on their distance to v then according to the indices of their labels, and pick out the first κ_v of them—which can be done with complexity $O(\kappa_v\varpi + |V|) \leq O(N\varpi + |V|)$. So *Operation* 2 has complexity $O(N|V| + |V|(N\varpi + |V|)) = O(N\varpi|V| + |V|^2)$.

Now consider *Operation* 3. For each vertex v, we sort the color-slots in S_v based on the indices of their labels, which has complexity $O(\kappa_v \log \kappa_v)$; then, if a color-slot s_i is the jth element in the sorted list, it means $|S_v \cap \{s_t | t \leq i\}| = j$, which can be used to update the value of X_i. So *Operation* 3 has complexity $O(\sum_{v\in V} \kappa_v \log \kappa_v) \leq O(|V|N \log N)$.

Operation 4 is similar to *Operation* 2, except that here we consider it for each color-slot s_i instead of for each vertex v. So *Operation* 4 can be seen to have complexity $O(\varpi(N\varpi + |V|)) = O(N\varpi^2 + \varpi|V|)$.

We can rightly assume that $\varpi \geq N$, because otherwise the data-interleaving problem would be trivial—we just make all the color-slots have different colors. Therefore, the total time complexity of the data-interleaving algorithm is $O(|V|^2) + O(\varpi + |V|) + O(N\varpi|V| + |V|^2) + O(|V|N \log N) + O(N\varpi^2 + \varpi|V|) = O(|V|^2 + N\varpi^2 + N\varpi|V|)$.

REFERENCES

Borodin, A. and El-Yaniv, R. 1998. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, U.K.

Dowdy, L. W. and Foster, D. V. 1982. Comparative models of the file assignment problem. *ACM Comput. Surv. 14*, 2, 287–313.

Garey, M. R. and Johnson, D. S. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company, New York, NY, and San Francisco, CA.

Hochbaum, D. S. and Shmoys, D. B. 1985. A best possible heuristic for the k-center problem. *Math. Operat. Res. 10*, 2, 180–184.

Jiang, A. and Bruck, J. 2003a. Diversity coloring for distributed data storage in networks. Unpublished manuscript.

Jiang, A. and Bruck, J. 2003b. Memory allocation in information storage networks. In *Proceedings of the IEEE International Symposium on Information Theory*, (Tokohama, Japan). 453.

Jiang, A. and Bruck, J. 2005. Multicluster interleaving on linear paths and cycles. *IEEE Trans. Inform. Theory 51*, 2 (Feb.), 597–611.

Jiang, A., Cook, M., and Bruck, J. 2004. Optimal t-interleaving on tori. In *Proceedings of the IEEE International Symposium on Information Theory* (Chicago, IL). 22.

Kalpakis, K., Dasgupta, K., and Wolfson, O. 2001. Optimal placement of replicas in trees with read, write, and storage costs. *IEEE Trans. Parallel Distrib. Syst. 12*, 6 (June), 628–637.

Kariv, O. and Hakimi, S. L. 1979. An algorithmic approach to network location problems. I: The p-centers. *SIAM J. Appl. Math. 37*, 3 (Dec.), 513–538.

Mahmoud, S. and Riordan, J. S. 1976. Optimal allocation of resources in distributed information networks. *ACM Trans. Database Syst. 1*, 66–78.

Malluhi, Q. M. and Johnston, W. E. 1998. Coding for high availability of a distributed-parallel storage system. *IEEE Trans. Parallel Distrib. Syst. 9*, 12 (Dec.), 1237–1252.

NAOR, M. AND ROTH, R. M. 1995. Optimal file sharing in distributed networks. *SIAM J. Comput. 24*, 1, 158–183.

PATTERSON, D. A., GIBSON, G. A., AND KATZ, R. 1988. A case for redundant arrays of inexpensive disks. In *Proceedings of the SIGMOD International Conference on Data Management*. 109–116.

SLATER, P. J. 1976. R-domination in graphs. *J. Assoc. Comput. Mach. 23*, 3 (March), 446–450.

WANG, J. 1999. A survey of Web caching schemes for the Internet. *ACM SIGCOMM Comput. Comm. Rev. 29*, 5, 36–46.

Received August 2004; revised September 2004; accepted September 2004