Memory Allocation In Information Storage Networks[1]

Anxiao (Andrew) Jiang and Jehoshua Bruck
California Institute of Technology
Pasadena, CA 91125, U.S.A.
{jax, bruck}@paradise.caltech.edu

Abstract — We propose a file storage scheme which bounds the file-retrieving delays in a heterogeneous information network, under both fault-free and faulty circumstances. The scheme combines coding with storage for better performance. We study the memory allocation problem for the scheme, which is to decide how much data to store on each node, with the objective of minimizing the total amount of data stored in the network. This problem is NP-hard for general networks. We present three polynomial-time algorithms which solve the memory allocation problem for tree networks. The first two algorithms are for tree networks with and without upper bounds on nodes' memory sizes respectively. The third algorithm finds, among all the optimal solutions for the tree network, the solution that minimizes the greatest memory size of single nodes. By combining these memory allocation algorithms with known data-interleaving techniques, a complete solution to realize the file storage scheme in tree networks is established.

Index Terms — Distributed networks, domination, error-correcting codes, file assignment, memory allocation, NP-hard, quality of service (QoS), tree.

I. Introduction

Given a graph $G(V,E)$, where each edge $e \in E$ has a length $l(e)$, we define $d(u,v)$ as the length of the shortest path between vertex $u \in V$ and $v \in V$, and call it the 'distance between u and v'. For a vertex $u \in V$ and a real number r, we define $N(u,r)$ as the set of vertices within distance r from u, namely, $N(u,r) = \{v | v \in V, d(u,v) \leq r\}$.

We study the following problem in this paper:

Definition 1: The Memory Allocation Problem

INSTANCE: A graph $G = (V,E)$. Every edge $e \in E$ has a length $l(e)$. 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 parameter $W_{min}(v)$, which is called the '*minimum memory size*' of v, and a parameter $W_{max}(v)$, which is called the '*maximum memory size*' of v.

QUESTION: How to assign a number $w(v)$ ($W_{min}(v) \leq w(v) \leq W_{max}(v)$) to each vertex $v \in V$, with the value of $\sum_{v\in V} w(v)$ minimized, such that for any vertex $u \in V$ and for $1 \leq i \leq n_u$, $\sum_{v \in N(u, r_i(u))} w(v) \geq k_i(u)$? (Here $w(v)$ is called the '*memory size* of v'. A solution to this memory allocation problem is called an *optimal memory allocation.*)

COMMENTS: Each element in a 'requirement set' $R(v)$ is a pair of numbers, written in the form as (r,k). In the definition of this problem, both $l(e)$ and $r_i(v)$ are non-negative real numbers, and $k_i(v)$, $W_{min}(v)$, $W_{max}(v)$, and $w(v)$ are all non-negative integers. □

We use the following example to illustrate the memory allocation problem and reveal its applications.

Example 1: Fig. 1 shows an example of the memory allocation problem. The graph $G = (V,E)$ has 6 vertices; the number beside each edge is its length. A solution is also given: $w(v_1) = 3$, $w(v_2) = 2$, $w(v_3) = 2$, $w(v_4) = 5$, $w(v_5) = 7$, $w(v_6) = 3$, which is shown in the figure. We claim without proof that no other solution has a smaller value of $\sum_{v\in V} w(v)$; readers can verify that the claim is true.

The memory allocation problem has applications in data storage. The distance between two nodes $u \in V$ and $v \in V$ can be seen as the delay of transmitting data between those two nodes. And the requirement set of a node v shows v's requirements on data-retrieving delays: if $(r,k) \in R(v)$, then the maximum delay that v can tolerate in order to retrieve k units of data from the network equals r. $W_{max}(v)$ and $W_{min}(v)$ are the

[1]This work was supported in part by the Lee Center for Advanced Networking at the California Institute of Technology.

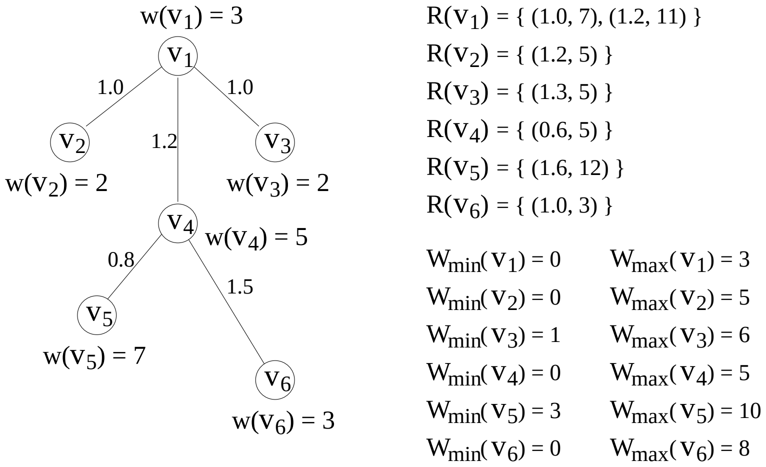

Figure 1: An example of the memory allocation problem.

upper and lower bounds on the amount of data that can be stored on node v, and $w(v)$ is the actual number of units of data stored on v.

As an example, with the solution in Fig.1, the delays for v_1 to retrieve 7 and 11 units of data are 1.0 and 1.2 respectively; the parameters in $R(v_1)$ also guarantees the following fault-tolerant performance: when no more than $11 - 7 = 4$ units of data stored on nodes in $N(v_1, 1.2)$ become inaccessible, the delay for v_1 to retrieve 7 units of data will be no more than 1.2.

The memory allocation problem can also have many other applications. For example, the graph can represent a transport network, and gasoline, food or medicine is stored on nodes. Then a solution to the problem will bound the delays to transport certain amounts of the corresponding substance to every node. □

The above example reveals that the memory allocation problem is related to data storage, but exactly how it can be used for storing files in information networks is not shown. This problem is in fact a sub-problem of a file storage scheme we propose. The scheme bounds the file-retrieving delays in a heterogeneous information network, under both fault-free and faulty circumstances. And it combines coding with storage for better performance. The scheme will be introduced in Section II.

The memory allocation problem is NP-hard for general graphs because the NP-complete dominating set problem [1] can be reduced to it. In this paper we study the memory allocation problem for trees, and present three polynomial-time algorithms. The first two algorithms solve the memory allocation problem for trees with and without upper bounds on nodes' memory sizes respectively. They have complexities of $O(q|V|^3)$ and $O(q|V|^2)$, where q is the average cardinality of a requirement set. The third algorithm finds a solution that minimizes the greatest memory size of single nodes, among all the optimal solutions for the tree. It's complexity is $O(q|V|^3 log(Y - \frac{X}{|V|}))$, where Y is the greatest memory size of single nodes in some memory-allocation solution, and X is the total memory size in that solution.

II. Background

Research in file storage is becoming more active today with the appearing of more large information networks such as the Internet. For popular public files, many schemes have been proposed to distributively store replications of the files in different parts of the network, in order to shorten file-retrieving delay, reduce data flow, balance load and improve data availability. Combining coding with storage can yield significantly better performance. In [2] and [3] such schemes have been studied, where a file is encoded into a codeword using an error-correcting code, and components of the codeword are distributively stored in such a way that for every network node, it can retrieve enough components for file-recovering from memories within a fixed number of hops. Both schemes aim to bound data-retrieving delay for quality-of-service (QoS). In those schemes, the file is seen as a string of k symbols, and is encoded using an (n, k) code to get a total of n $(n \geq k)$ symbols. Say the code can correct up to ε erasures, then any $n - \varepsilon$ distinct symbols of the codeword are sufficient for recovering the file. If an MDS code is used, then $n - \varepsilon = k$.

We propose a scheme which generalizes the schemes in [2] and [3] in the following ways. Each network link is assigned a length which represents the delay of transmitting the file over that link. Then the delay within which a node can retrieve enough data for file-recovering, called the file-retrieving delay, equals the distance from the node within which $n - \varepsilon$ distinct codeword symbols are stored. We allow different nodes to have different upper bounds on the file-retrieving delays, instead of having one uniform upper bound as in [2] [3]. What's more, it's desirable that the number of distinct symbols within a certain distance from a node grows steadily when that distance increases — so that the file-retrieving delay will degrade gracefully when more and more symbols become inaccessible (e.g., because of data loss or busy processors) — and we allow different nodes to have different specifications on such a relationship. We also allow each node to have an upper bound and a lower bound on the numbers of symbols it can store. (The lower bound can exist for various reasons, e.g., a web server typically always stores a copy of each file it generates.) By representing the network with a graph whose vertices and edges correspond to network nodes and links respectively, the scheme is formally defined as follows.

Definition 2 (The Generalized File Storage Scheme) : The file is encoded into a codeword of n symbols, any $n - \varepsilon$ of which are sufficient for recovering the file. The network is modelled as a graph $G = (V, E)$, where each edge $e \in E$ has a length $l(e)$. Each vertex $v \in V$ is associated with a set $R(v) = \{(r_i(v), k_i(v)) | 1 \leq i \leq n_v\}$, called its '*requirement set*', a '*minimum memory size*' $W_{min}(v)$, and a '*maximum memory size*' $W_{max}(v)$.

The file storage scheme is to assign $w(v)$ $(W_{min}(v) \leq w(v) \leq W_{max}(v))$ codeword symbols to each vertex $v \in V$, such that for any vertex $u \in V$ and for $1 \leq i \leq n_u$, vertices in $N(u, r_i(u))$ store at least k_i distinct symbols. Among all the solutions that satisfy the above conditions, the one with the minimum value of $\sum_{v \in V} w(v)$ is called *optimal.* □

The above scheme aims to guarantee quality-of-service (QoS) in both fault-free and faulty circumstances. Each node can bound the file-retrieving delays at different degrees of data inaccessibility by appropriately setting values in its 'requirement set'. The scheme allows each node to have a separate requirement set, to better accommodate the fact that in real networks, nodes typically have quite diverse QoS requirements.

Finding an optimal solution to the above scheme has two steps: deciding how many symbols to assign to each vertex, called 'memory allocation', and deciding which symbols to assign to each vertex, called 'symbol mapping'. Usually these two steps are dependent on each other. But if n, the total number of symbols, is sufficiently large, then 'memory allocation' becomes independent of 'symbol mapping', and becomes the problem defined in Definition 1. Having n sufficiently large to make the memory allocation and the symbol mapping independent doesn't necessarily mean that n has to be very large. For example, it can be shown that for tree networks, those two problems are always independent regardless of how large n is. If n is not sufficiently large, a solution to the memory allocation problem will still provide us with a lower bound on the amount of data to be stored in the network for the file storage scheme.

III. Memory Allocation for Trees

Trees are important network structures. For example, trees are often used by backbone networks, by virtual private networks (VPNs) [4], and as embedded networks. In the remaining of this paper, we'll purely focus on the memory allocation problem for a tree $G(V, E)$.

The following proposition is self-evident.

Proposition 1 *The memory allocation problem has a solution if and only if for any $v \in V$ and for $1 \leq i \leq n_v$, $\sum_{u \in N(v, r_i(v))} W_{max}(u) \geq k_i(v)$.* □

From now on we always assume a solution exists for the memory allocation problem.

For any two vertices v_1 and v_2 in a tree $G = (V, E)$, 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 between v_1 and the root. 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.

Definition 3 (An Optimal Memory Basis) : A set $\{w(v)|v \in V\}$ is called an *optimal memory basis* if there exists an optimal memory allocation for the tree $G = (V, E)$ which assigns memory size $w_{opt}(v)$ to every vertex $v \in V$, such that for any $v \in V$, $W_{min}(v) \leq w(v) \leq w_{opt}(v)$. □

The following lemma shows how one can get a new optimal memory basis from an old optimal memory basis by increasing memory sizes.

Lemma 1 *u_1 is a child of u_2 in the tree $G = (V, E)$. And $\{w_1(v)|v \in V\}$ is an optimal memory basis. Assume the following conditions are true for the memory allocation problem: for any vertex $v \in Des(u_1)$, the 'requirement set' $R(v) = \emptyset$; $R(u_1)$ has an element (r, k), namely, $(r, k) \in R(u_1)$.*

We define S_1 as $S_1 = N(u_2, r - d(u_1, u_2))$, and define S_2 as $S_2 = N(u_1, r) - S_1$. We compute the elements of a set $\{w_2(v)|v \in V\}$ in the following way (step 1 to step 3):

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

Step 2: Let

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

, and let $C \leftarrow S_2$.

Step 3: Let v_0 be the vertex in C that is the closest to u_1—namely, $d(v_0, u_1) = \min_{v \in C} d(v, u_1)$. 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 conclusion is true: $\{w_2(v)|v \in V\}$ is also an optimal memory basis. □

The proof of Lemma 1 is presented in Appendix I for interested readers.

The following lemma shows how one can transform one memory allocation problem into another by modifying the 'requirement sets' and the 'minimum memory sizes'.

Lemma 2 *u_1 is a child of u_2 in the tree $G = (V, E)$. And $\{w_0(v)|v \in V\}$ is an optimal memory basis. Assume the following conditions are true for the memory allocation problem: for any vertex $v \in Des(u_1)$, the 'requirement set' $R(v) = \emptyset$; for any element in $R(u_1)$—say the element is (r, k)—we have $\sum_{u \in N(u_2, r - d(u_1, u_2))} W_{max}(u) + \sum_{u \in N(u_1, r) - N(u_2, r - d(u_1, u_2))} w_0(u) \geq k$.*

We compute the elements of a set $\{\hat{R}(v)|v \in V\}$ in the following way (step 1 and step 2):

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 $\sum_{v \in N(u_1, r)} w_0(v) < k$, then add an element $(r - d(u_1, u_2), k - \sum_{v \in N(u_1, r) - N(u_2, r - d(u_1, u_2))} w_0(v))$ 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 'minimum memory size' is $w_0(v)$ instead of $W_{min}(v)$.*

Then the following conclusions are true:

(1) The 'new problem' *has a solution (an optimal memory allocation).*

(2) An optimal memory allocation for the 'new problem' *is also an optimal memory allocation for the* 'old problem'. □

The proof of Lemma 2 is presented in Appendix II.

Based on the above two lemmas, we naturally get the following memory allocation algorithm for trees. The algorithm processes all the vertices one by one. Every time a vertex is processed, it uses the method in Lemma 1 to update the memory sizes , and uses the method in Lemma 2 to transform the memory allocation problem, until a solution is found.

Algorithm 1 [Memory Allocation on a Tree]

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 any of its ancestors. For each vertex $\tilde{v} \in V$ that is not the root, it is processed with the following two steps:

Step 1: Treat $\tilde{v}$, the parent of $\tilde{v}$ and the set $\{w(v)|v \in V\}$ as the vertex 'u_1', the vertex 'u_2' and the set '$\{w_1(v)|v \in V\}$' in Lemma 1 respectively, and for each element in $R(\tilde{v})$ do the following two things: (1) treat this element in $R(\tilde{v})$ as the element '(r,k)' in Lemma 1, and compute the set '$\{w_2(v)|v \in V\}$' as in Lemma 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 $\{w(v)|v \in V\}$ as the vertex 'u_1', the vertex 'u_2' and the set '$\{w_0(v)|v \in V\}$' in Lemma 2 respectively, and do the following two things: (1) compute the set '$\{\hat{R}(v)|v \in V\}$' as in Lemma 2; (2) for every vertex $v \in V$, let $R(v) \leftarrow \hat{R}(v)$, and let $W_{min}(v) \leftarrow w(v)$.

Denote the root by v_{root}. The vertex v_{root} is processed in the following way:

Pretend that the root v_{root} has a parent that is infinitely far away. Treat v_{root}, the parent of v_{root} and the set $\{w(v)|v \in V\}$ as the vertex 'u_1', the vertex 'u_2' and the set '$\{w_1(v)|v \in V\}$' in Lemma 1 respectively, and for each element in $R(v_{root})$ do the following two things: (1) treat this element in $R(v_{root})$ as the element '(r,k)' in Lemma 1, and compute the set '$\{w_2(v)|v \in V\}$' as in Lemma 1; (2) for every vertex $v \in V$, let $w(v)$ get the value of $w_2(v)$ — namely, $w(v) \leftarrow w_2(v)$.

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 pseudo-code of Algorithm 1 is presented in Appendix III for interested readers.

Analysis shows that Algorithm 1 has 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 1 are presented in Appendix IV.

IV. Memory Allocation for Trees without Upper Bounds on Memory Sizes

In the memory allocation problem, every vertex $v \in V$ has a 'maximum memory size' $W_{max}(v)$. If for every $v \in V$, $W_{max}(v)$ is infinitely large, then we say that there are no upper bounds on memory sizes. For such a special case, some techniques can be used to get a memory allocation algorithm for trees of complexity less than $O(q|V|^3)$, which we will present in this section.

The new algorithm is very similar to Algorithm 1, except that in this new algorithm, a new notion named '*residual requirement set*' is adopted. 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 as follows: $\bar{r} = r$, and $\bar{k} = \max\{k - \sum_{u \in N(v,r)} w(u), 0\}$. (The meaning of the element $(\bar{r},\bar{k})$ is that the memories of the nodes in $N(v,r)$ needs to be increased by $\bar{k}$ so that $\sum_{u \in N(v,r)} w(u)$ will be no less than k.)

We use '*Algorithm 2*' to denote the new algorithm which finds solutions to the memory allocation problem for trees without upper bounds on memory sizes. For simplicity of this paper, we omit the presentation of the actual algorithm, which has a similar structure as Algorithm 1. We present the pseudo-code of Algorithm 2 in Appendix V for interested readers.

The complexity of Algorithm 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 are no upper bounds on 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 2 is $O(q|V|^2)$.

V. Minimizing the Greatest Memory Size of Single Nodes

Minimizing the maximum amount of resource assigned to a single place often has engineering importance in resource assignment problems. In this section, we present an algorithm which finds, among all the solutions to the memory allocation problem for a tree, a solution that minimizes the greatest memory size of single nodes. That is, the algorithm finds an 'optimal memory allocation' whose value of $\max_{v \in V} w(v)$ is minimized.

Algorithm 3 [Memory Allocation on a Tree with Minimized Greatest Memory Size]

1. Use Algorithm 1 to find an optimal memory allocation. Say the optimal memory allocation assigns memory size $w_{opt}(v)$ to each vertex $v \in V$. We let $X \leftarrow \sum_{v \in V} w_{opt}(v)$, and let $Y \leftarrow \max_{v \in V} w_{opt}(v)$.

In this algorithm we use 'L' and 'U' to represent the lower limit and the upper limit for the minimized greatest memory size of single nodes; and we use '$M_{min-max}$' to denote the minimized greatest memory size (which is unknown yet). Since every optimal memory allocation's total memory size of all vertices equals X, $M_{min-max}$ must be no less than $\lceil \frac{X}{|V|} \rceil$. Since we've already found an optimal memory allocation whose greatest memory size is Y, $W_{min-max}$ must be no greater than Y. So initially, we let $L \leftarrow \lceil \frac{X}{|V|} \rceil$, and let $U \leftarrow Y$.

2. Use a binary search to find out the exactly value of $W_{min-max}$ in the following way. Let $m \leftarrow \lfloor \frac{L+U}{2} \rfloor$. Use Algorithm 1 to find an optimal memory allocation for such a memory allocation problem: everything in this problem is the same as in the original memory allocation problem, except that in this problem, the 'maximum memory size' of each vertex $v \in V$ is $\min\{m, W_{min}(v)\}$ instead of $W_{min}(v)$. If this problem has a solution and in the solution the total memory size equals X, then it means that $M_{min-max}$ is no greater than m, so we let $U \leftarrow m$; otherwise, it means that $M_{min-max}$ is greater than m, so we let $L \leftarrow m+1$. Repeat this procedure until L and U become equal.

Now we have $W_{min-max} = L = U$. Use Algorithm 1 to find a solution to the following memory allocation problem: in the problem everything is the same as in the original memory allocation problem, except that in this problem, the 'maximum memory size' of each vertex $v \in V$ is $\min\{W_{min-max}, W_{min}(v)\}$ instead of $W_{min}(v)$. The solution found is the solution whose greatest memory size is minimized.

□

The binary search has $O(\log(Y - \lceil \frac{X}{|V|} \rceil))$ steps; in every step Algorithm 1 is executed once. So Algorithm 3 has complexity $O(q|V|^3 \log(Y - \frac{X}{|V|}))$. (To see how large Y can be, note that Y is never greater than $\max_{v \in V, 1 \leq i \leq n_v} k_i(v)$ or $\max_{v \in V} W_{max}(v)$.)

[**Concluding Remarks**] It can be shown that for a tree network, with a memory allocation generated by the algorithms in this paper, there always exists a way to map file symbols to the memories to realize the file storage scheme proposed in Definition 2. The 'symbol mapping' uses a data-interleaving technique which is a generalization of the result in [3]. So by combining the memory allocation algorithms presented in this paper with the developed data-interleaving technique, a complete solution to realize the proposed file storage scheme in tree networks is established.

References

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

[2] M. Naor and R. M. Roth, "Optimal file sharing in distributed networks," SIAM J. Comput., vol. 24, no. 1, pp. 158–183, 1995.

[3] A. Jiang and J. Bruck, "Diversity Coloring for Information Storage in Networks," in Proceedings of the 2002 IEEE International Symposium on Information Theory, Lausanne, Switzerland, 2002, pp. 381.

[4] A. Kumar, R. Rastogi, A. Silberschatz and B. Yener, "Algorithms for Provisioning Virtual Private Networks in the Hose Model," IEEE/ACM Transactions on Networking, vol. 10, no. 4, pp. 565–578, 2002.

Appendix

APPENDIX I

In this appendix, we present the proof of Lemma 1.

Proof: The following three conclusions can be easily seen to be true:

Conclusion (1): $S_1 \cup S_2 = N(u_1, r)$, and $S_1 \cap S_2 = \emptyset$.

Conclusion (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

$$\sum_{v \in V} w_2(v) - \sum_{v \in V} w_1(v) = \max\{0, k - \sum_{v \in S_1} W_{max}(v) - \sum_{v \in S_2} w_1(v)\}$$

Conclusion (3):

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

Now let's use them to prove Lemma 1.

$\{w_1(v) | v \in V\}$ is an optimal memory basis. So there exists an optimal memory allocation that assigns memory size $w_{opt}(v)$ to every vertex $v \in V$, such that $w_1(v) \leq w_{opt}(v)$ for any $v \in V$. By Definition 1 we know $\sum_{v \in N(u_1,r)} w_{opt}(v) \geq k$. Since $\sum_{v \in N(u_1,r)} w_{opt}(v) = \sum_{v \in S_1} w_{opt}(v) + \sum_{v \in S_2} w_{opt}(v) \leq \sum_{v \in S_1} W_{max}(v) + \sum_{v \in S_2} w_{opt}(v)$, we get $\sum_{v \in S_2} w_{opt}(v) \geq k - \sum_{v \in S_1} W_{max}(v)$. And clearly $\sum_{v \in S_2} w_{opt}(v) \geq \sum_{v \in S_2} w_1(v)$. By conclusion (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 $\sum_{v \in S_2} w_{opt}(v) \geq \sum_{v \in S_2} w_2(v)$.

We compute the elements of a set $\{w_o(v) | v \in V\}$ in the following way (step 1 to step 3):

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, $d(v_0, u_1) = \min_{v \in C} d(v, u_1)$. 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 X equals 0.

From the above three steps, it's 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's easy 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 $d(v, u_1) < d(v_1, u_1)$, $w_o(v) = W_{max}(v)$; 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 $d(v, u_1) > d(v_2, u_1)$, $w_o(v) = w_1(v)$. Therefore for any real number L, if we define Q as $Q = \{v | v \in S_2, d(v, u_1) \leq L)\}$, 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 $S_2 \subseteq Des(u_1) \cup \{u_1\}$, $N(v_0, r_0) = \{v | v \in N(v_0, r_0), v \notin S_2\} \cup \{v | v \in N(v_0, r_0), v \in S_2\} = \{v | v \in N(v_0, r_0), v \notin S_2\} \cup \{v | v \in S_2, d(v, v_0) \leq r_0\} = \{v | v \in N(v_0, r_0), v \notin S_2\} \cup \{v | v \in S_2, d(v, u_1) \leq r_0 - d(u_1, v_0)\}$. So $\sum_{v \in N(v_0, r_0)} w_o(v) \geq \sum_{v \in N(v_0, r_0)} w_{opt}(v)$. Clearly $\sum_{v \in N(v_0, r_0)} w_{opt}(v) \geq k_0$. So $\sum_{v \in N(v_0, r_0)} w_o(v) \geq k_0$. Since $\sum_{v \in V} w_o(v) = \sum_{v \in V} w_{opt}(v)$, the memory allocation which assigns memory size $w_o(v)$ to every vertex $v \in V$ is also an optimal memory allocation.

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.

□

APPENDIX II

In this appendix, we present the proof of Lemma 2.

Proof: Conclusion (1) can be easily proved by using Proposition 1 and the assumption of this paper that the *'old problem'* has a solution. Below we give the proof of conclusion (2).

Consider an optimal memory allocation for the *'new problem'* which 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 $\sum_{u \in N(\bar{v}, \bar{r})} \hat{w}_{opt}(u) \geq \bar{k}$. Now consider the case where $(\bar{r}, \bar{k}) \notin \hat{R}(\bar{v})$. Clearly in this case $\bar{v} = u_1$, and either $\sum_{u \in N(u_1, \bar{r})} w_0(u) \geq \bar{k}$, or $\sum_{u \in N(u_1, \bar{r})} w_0(u) < \bar{k}$. If $\sum_{u \in N(u_1, \bar{r})} w_0(u) \geq \bar{k}$, since $\hat{w}_{opt}(u) \geq w_0(u)$ for any $u \in V$, we have $\sum_{u \in N(\bar{v}, \bar{r})} \hat{w}_{opt}(u) \geq \bar{k}$. We define S_1 as $S_1 = N(u_2, \bar{r} - d(u_1, u_2))$, and define S_2 as $S_2 = N(u_1, \bar{r}) - S_1$. Then if $\sum_{u \in N(u_1, \bar{r})} w_0(u) < \bar{k}$, its simple to see that $(\bar{r} - d(u_1, u_2), \bar{k} - \sum_{u \in S_2} w_0(u)) \in \hat{R}(u_2)$. So $\sum_{u \in N(\bar{v}, \bar{r})} \hat{w}_{opt}(u) = \sum_{u \in S_1} \hat{w}_{opt}(u) + \sum_{u \in S_2} \hat{w}_{opt}(u) \geq \bar{k} - \sum_{u \in S_2} w_0(u) + \sum_{u \in S_2} \hat{w}_{opt}(u) \geq \bar{k}$. Therefore $\sum_{u \in N(\bar{v}, \bar{r})} \hat{w}_{opt}(u) \geq \bar{k}$ in all cases.

$\{w_0(v) | v \in V\}$ is an optimal memory basis for the *'old problem'*. So there exists an optimal memory allocation for the '*old problem*' which 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 $\{w_4(v) | v \in V\}$ — in the following way (step 1 to step 5):

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

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

Step 3: for any $v \in V$, 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, $d(v_0, u_2) = \min_{v \in C} d(v, u_2)$. 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 any $v \in V$, let $w_4(v) \leftarrow w_1(v) + w_3(v)$.

From the above five steps, it's simple to see that the following must be true: $\sum_{v \in V} w_{opt}(v) = \sum_{v \in V} w_1(v) + \sum_{v \in V} w_2(v) = \sum_{v \in V} w_4(v)$, 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, $\sum_{v \in N(u_2, L)} w_3(v) \geq \sum_{v \in N(u_2, L)} w_2(v)$; for any $v \in V$, if $w_4(v) < W_{max}(v)$, then $\sum_{u \in N(u_2, d(v, u_2))} w_3(u) = \sum_{u \in V} w_2(u)$.

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 $\sum_{v \in N(\hat{v}, \hat{r})} w_4(v) = \sum_{v \in N(\hat{v}, \hat{r})} w_1(v) + \sum_{v \in N(\hat{v}, \hat{r})} w_3(v) \geq \sum_{v \in N(\hat{v}, \hat{r})} w_1(v) + \sum_{v \in N(u_2, \hat{r} - d(\hat{v}, u_2))} w_3(v) \geq \sum_{v \in N(\hat{v}, \hat{r})} w_1(v) + \sum_{v \in N(u_2, \hat{r} - d(\hat{v}, u_2))} w_2(v) = \sum_{v \in N(\hat{v}, \hat{r})} w_1(v) + \sum_{v \in N(\hat{v}, \hat{r})} w_2(v) = \sum_{v \in N(\hat{v}, \hat{r})} w_{opt}(v) \geq \hat{k}$.

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

So $\sum_{v \in N(\hat{v}, \hat{r})} w_4(v) \geq \hat{k}$ in all cases. So $\sum_{v \in V} \hat{w}_{opt}(v) \leq \sum_{v \in V} w_4(v)$. Since we also have $\sum_{v \in V} w_4(v) = \sum_{v \in V} w_{opt}(v)$ and $\sum_{v \in V} w_{opt}(v) \leq \sum_{v \in V} \hat{w}_{opt}(v)$, we get $\sum_{v \in V} \hat{w}_{opt}(v) = \sum_{v \in V} w_{opt}(v)$. So the optimal memory allocation for the *'new problem'*, which assigns 'memory size' $\hat{w}_{opt}(v)$ to every vertex $v \in V$, is also an optimal memory allocation for the *'old problem'*. So conclusion (2) is proved.

□

APPENDIX III

In this appendix we present the pseudo-code of Algorithm 1.

Algorithm 1 [Memory Allocation on a Tree]

1. Label the vertices in V as v_1, v_2, $\cdots$, $v_{|V|}$ according to the following rule: if v_i is an ancestor of v_j, then $i > j$. Let $w(v_i) \leftarrow W_{min}(v_i)$ for $1 \leq i \leq |V|$.

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 $S_1 = N(v_P, r - d(v_i, v_P))$, and define S_2 as $S_2 = N(v_i, r) - S_1$. Update the elements in $\{w(v)|v \in V\}$ in the following way (step 1 and step 2):

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, $d(u_0, v_i) = \min_{u \in C} d(u, v_i)$. 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 $\sum_{u \in N(v_i, r)} w(u) < k$, then add an element $(r - d(v_i, v_P), k - \sum_{u \in N(v_i, r) - N(v_P, r - d(v_i, v_P))} w(u))$ 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 $\{w(v)|v \in V\}$ in the following way (step 1 and step 2):

Step 1: Let

$$X \leftarrow \max\{0, k - \sum_{u \in N(v_{|V|}, r)} w(v)\}$$

, and let $C \leftarrow V$.

Step 2: Let u_0 be the vertex in C that is the closest to $v_{|V|}$—namely, $d(u_0, v_{|V|}) = \min_{u \in C} d(u, v_{|V|})$. 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 $w(v_1)$, $w(v_2)$, $\cdots$, $w(v_{|V|})$ as the solution to the memory allocation problem.

□

Note that in the above pseudo-code, the values of 'minimum memory sizes' are not really updated because it's not necessary to do that, even though it has been used in other parts of the paper as a helpful tool for analysis.

APPENDIX IV

In this appendix, we prove the correctness of Algorithm 1, and analyze its complexity.

Theorem 1 *Algorithm 1 is correct.*

Proof: At the beginning of Algorithm 1, the value of each $w(v)$ $(v \in V)$ is set to be $W_{min}(v)$. Clearly at this moment, $\{w(v)|v \in V\} = \{W_{min}(v)|v \in V\}$ is an 'optimal memory basis'.

Then Algorithm 1 processes all the vertices one by one. Each vertex — including the root, in fact — is processed with the following two steps:

Step 1: Modify the value of the set $\{w(v)|v \in V\}$, using the method in Lemma 1.

Step 2: Modify the values of $\{R(v)|v \in V\}$ and $\{W_{min}(v)|v \in V\}$, using the method in Lemma 2. (Therefore the parameters in the memory allocation problem are changed. Using the terms in Lemma 2, the memory allocation problem is changed from an '*old problem*' to a '*new problem*'.)

It's easy to prove by induction that the following two assertions are true:

Assertion 1: Every time a vertex is processed, after step 1, the set $\{w(v)|v \in V\}$ is still an 'optimal memory basis'.

Assertion 2: Every time a vertex is processed, after step 2, the '*new problem*' still has an solution, and any solution of the '*new problem*' is also a solution of the original memory allocation problem.

When all the vertices are processed, all the 'requirement sets' become empty sets, so at this moment the 'optimal memory basis', which is $\{w(v)|v \in V\}$, is also an optimal memory allocation. So Algorithm 1 finds the correct solution.

□

Complexity analysis: Algorithm 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 in 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 1 is $O(|V|^2 + |V|^2 + q|V|^2 \cdot |V|)$, which equals $O(q|V|^3)$.

APPENDIX V

In this appendix we present the pseudo-code of Algorithm 2.

Algorithm 2 [Memory Allocation on a Tree without Upper Bounds on Memory Sizes]

1. Label the vertices in V as $v_1, v_2, \cdots, v_{|V|}$ according to the following rule: if v_i is an ancestor 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 $k - \sum_{v \in N(v_i,r)} w(v) > 0$, then let $Res(v_i) \leftarrow Res(v_i) \cup \{(r, k - \sum_{v \in N(v_i,r)} w(v))\}$.

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 $Q(v_i) \neq \emptyset$ do:

{ Let (r,k) be any element in $Q(v_i)$. If $r < d(v_i, v_P)$, 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 $r \ge d(v_i, v_j)$, then let $(r,k) \leftarrow (r, k-x)$; if $k \le 0$, 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 $Res(v_P) \leftarrow Res(v_P) \cup \{(r - d(v_i, v_P), k - x)\}$.

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 $w(v_1)$, $w(v_2)$, $\cdots$, $w(v_{|V|})$ as the solution to the memory allocation problem.

$\square$