Efficient Algorithms for the Ring Loading Problem with Demand Splitting

Biing-Feng Wang, Yong-Hsian Hsieh, and Li-Pu Yeh

Department of Computer Science, National Tsing Hua University Hsinchu, Taiwan 30043, Republic of China,
bfwang@cs.nthu.edu.tw, {eric,lee}@venus.cs.nthu.edu.tw
Fax: 886-3-5723694

Abstract. Given a ring of size n and a set K of traffic demands, the ring loading problem with demand splitting (RLPW) is to determine a routing to minimize the maximum load on the edges. In the problem, a demand between two nodes can be split into two flows and then be routed along the ring in different directions. If the two flows obtained by splitting a demand are restricted to integers, this restricted version is called the ring loading problem with integer demand splitting (RLPWI). In this paper, efficient algorithms are proposed for the RLPW and the RLPWI. Both the proposed algorithms require $O(|K| + t_s)$ time, where t_s is the time for sorting $|K|$ nodes. If $|K| \geq n^{\epsilon}$ for some small constant $\epsilon > 0$, integer sort can be applied and thus $t_s = O(|K|)$; otherwise, $t_s = O(|K| \log |K|)$. The proposed algorithms improve the previous upper bounds from $O(n|K|)$ for both problems.

Keywords: Optical networks, rings, routing, algorithms, disjoint-set data structures

1 Introduction

Let R be a ring network of size n, in which the node-set is $\{1, 2, \ldots, n\}$ and the edge-set is $E=\{(1, 2), (2, 3), \ldots, (n-1, n), (n, 1)\}$. Let K be a set of traffic demands, each of which is described by an origin-destination pair of nodes together with an integer specifying the amount of traffic requirement. The ring is undirected. Each demand can be routed along the ring in any of the two directions, clockwise and counterclockwise. A demand between two nodes i and j, where $i < j$, is routed in the *clockwise* direction if it passes through the node sequence $(i, i+1, \ldots, j)$, and is routed in the *counterclockwise* direction if it passes through the node sequence $(i, i-1, \ldots, 1, n, n-1, \ldots, j)$. The *load* of an edge is the total traffic flow passing through it. Given the ring-size n and the demand-set K, the *ring loading problem* (*RLP*) is to determine a routing to minimize the maximum load of the edges.

There are two kinds of RLP. If each demand in K must be routed entirely in either of the directions, the problem is called the *ring loading problem without demand splitting* (*RLPWO*). Otherwise, the problem is called the *ring loading*

G. Di Battista and U. Zwick (Eds.): ESA 2003, LNCS 2832, pp. 517–526, 2003.

problem with demand splitting (*RLPW*), in which each demand may be split between both directions. In RLPW, it is allowed to split a demand into two fractional flows. If the two flows obtained by splitting a demand are restricted to integers, this restricted version is called the *ring loading problem with integer demand splitting* (*RLPWI*).

The RLP arose in the planning of optical communication networks that use bi-directional SONET (Synchronous Optical Network) rings [2,7,8]. Because of its practical significance, many researchers have turned their attention to this problem. Cosares and Saniee [2] showed that the RLPWO is NP-hard if more than one demand is allowed between the same origin-destination pair and the demands can be routed in different directions. Two approximation algorithms had been presented for the RLPWO. One was presented by Schrijver, Seymour, and Winkler [7], which has a performance guarantee of 3/2. The other was presented by Amico, Labbe, and Maffioli [1], which has a performance guarantee of 2. For the RLPW, Schrijver, Seymour, and Winkler [7] had an $O(n^2|K|)$-time algorithm, Vachani, Shulman, and Kubat [8] had an $O(n^3)$-time algorithm, and Myung, Kim, and Tcha [5] had an $O(n|K|)$-time algorithm. For the RLPWI, Lee and Chang [4] had an approximation algorithm, Schrijver, Seymour, and Winkler [7] had an pseudo-polynomial algorithm, and Vachani, Shulman, and Kubat [8] had an $O(n^3)$-time algorithm. Very recently, Myung [6] gave some interesting properties for the RLPWI and proposed an $O(n|K|)$-time algorithm.

In this paper, efficient algorithms are proposed for the RLPW and the RLPWI. Both the proposed algorithms require $O(|K| + t_s)$ time, where t_s is the time for sorting $|K|$ nodes. If $|K| \geq n^\epsilon$ for some small constant $\epsilon > 0$, integer sort can be applied and thus $t_s = O(|K|)$; otherwise, $t_s = O(|K| \log |K|)$. For the real world application mentioned above, $|K|$ is usually not smaller than n and thus our algorithms achieve linear time. We remark that the problem size is $|K| + 1$ instead of $|K| + n$, since a ring can be simply specified by its size n. The proposed algorithms improve the previous upper bounds from $O(n|K|)$ for both problems. They are modified versions of the algorithms in [5,6].

For easy description, throughout the remainder of this paper, we assume that $2|K| \geq n$. In case this is not true, we transform in $O(t_s)$ time the given n and K into another instance n' and K' as follows. First, we sort the distinct nodes in K into an increasing sequence S. Then, we set $n' = |S|$ and replace each node in K by its rank in S to obtain K'.

The remainder of this paper is organized as follows. In the next section, notation and preliminary results are presented. Then, in Sections 3 and 4, $O(|K|+t_s)$-time algorithms are presented for the RLPW and the RLPWI, respectively. Finally, in Section 5, concluding remarks are given.

2 Notation and Preliminaries

Let $R = (V, E)$ be a ring of size n, where the node-set is $V = \{1, 2, \ldots, n\}$ and the edge-set is $E = \{(1, 2), (2, 3), \ldots, (n-1, n), (n, 1)\}$. For each i, $1 \leq i \leq n$, denote e_i as the edge $(i, (i \bmod n) +1)$. Let K be a set of demands. For easy

description, throughout this paper, the k-th demand in K is simply denoted by k, where $1 \leq k \leq |K|$. For each $k \in K$, let $o(k)$, $d(k)$, and $r(k)$ be, respectively, the origin node, the destination node, and the amount of traffic requirement, where $o(k) < d(k)$. Assume that no two demands have the same origin-destination pair; otherwise, we simply merge them into one. For each $k \in K$, let $E_k^+ = \{e_i | o(k) \leq i \leq d(k)-1\}$, which is the set of edges in the clockwise direction path from $o(k)$ to $d(k)$, and let $E_k^- = E \backslash E_k^+$, which is the set of edges in the counterclockwise direction path from $o(k)$ to $d(k)$. Let $\mathcal{X} = \{(x(1), x(2), \ldots, x(|K|)) | x(k)$ is a real number and $0 \leq x(k) \leq r(k)$ for each $k \in K\}$. Each $(x(1), x(2), \ldots, x(|K|)) \in \mathcal{X}$ defines a *routing* for K, in which for each $k \in K$ the flow routed clockwise is $x(k)$ and the flow routed counterclockwise is $r(k) - x(k)$. Given a routing X=($x(1)$, $x(2)$, …, $x(|K|)$), the *load* of each edge $e_i \in E$ is $g(X, e_i) = \sum_{k \in K, e_i \in E_k^+} x(k) + \sum_{k \in K, e_i \in E_k^-} (r(k) - x(k))$. The RLPW is to find a routing $X \in \mathcal{X}$ that minimizes $\max_{1 \leq i \leq n} g(X, e_i)$. The RLPWI is to find a routing $X \in \mathcal{X} \cap Z^{|K|}$ that minimizes $\max_{1 \leq i \leq n} g(X, e_i)$.

In the remainder of this section, some preliminary results are presented.

Lemma 1. *Given a routing $X = (x(1), x(2), \ldots, x(|K|))$, all $g(X, e_i), 1 \leq i \leq n$, can be computed in $O(|K|)$ time.*

Proof. We transform the computation into the problem of computing the prefix sums of a sequence of n numbers. First, we initialize a sequence ($s(1)$, $s(2)$, …, $s(n)$)=(0, 0, …, 0). Next, for each $k \in K$, we add $r(k) - x(k)$ to $s(1)$, add $-r(k) + 2x(k)$ to $s(o(k))$, and add $r(k) - 2x(k)$ to $s(d(k))$. Then, for $i = 1$ to n, we compute $g(X, e_i)$ as $s(1) + s(2) + \ldots + s(i)$. It is easy to check the correctness of the above computation. The lemma holds. □

Let $A = (a(1), a(2), \ldots, a(n))$ be a sequence of n values. The maximum of A is denoted by $max(A)$. The *suffix maximums* of A are elements of the sequence ($c(1)$, $c(2)$, …, $c(n)$) such that $c(i) = \max\{a(i), a(i+1), \ldots, a(n)\}$. For each i, $1 \leq i \leq n$, we define the function $\pi(A, i)$ to be the largest index $j \geq i$ such that $a(j) = \max\{a(i), a(i+1), \ldots, a(n)\}$. An element $a(j)$ is called a *suffix-maximum element* of A if $j = \pi(A, i)$ for some $i \leq j$. Clearly, the values of the suffix-maximum elements of A, from left to right, are strictly decreasing and the first such element is $max(A)$. Define $\Gamma(A)$ to be the index-sequence of the suffix-maximum elements of A. Let $\Gamma(A)$=($\gamma(1)$, $\gamma(2)$, …, $\gamma(q)$). According to the definitions of π and Γ, it is easy to see that $\pi(A, i)$=$\gamma(j)$ if and only if i is in the interval $[\gamma(j-1)+1, \gamma(j)]$, where $1 \leq j \leq q$ and $\gamma(0)$=0. According to the definition of suffix-maximum elements, it is not difficult to conclude the following two lemmas.

Lemma 2. *Let $S = (s(1), s(2), \ldots, s(l))$ and $T = (t(1), t(2), \ldots, t(m))$. Let $\Gamma(S) = (\alpha(1), \alpha(2), \ldots, \alpha(g))$ and $\Gamma(T) = (\beta(1), \beta(2), \ldots, \beta(h))$. Let $S \oplus T$ be the sequence $(s(1), s(2), \ldots, s(l), t(1), t(2), \ldots, t(m))$. If $s(\alpha(1)) \leq t(\beta(1))$, let $p = 0$; otherwise let p be the largest index such that $s(\alpha(p)) > t(\beta(1))$. Then, the sequence of suffix-maximum elements in $S \oplus T$ is $(s(\alpha(1)), s(\alpha(2)), \ldots, s(\alpha(p)), t(\beta(1)), t((2)), \ldots, t(\beta(h)))$.*

Lemma 3. *Let* $U = (u(1), u(2), \ldots, u(n))$ *and* $\Gamma(U) = (\gamma(1), \gamma(2), \ldots, \gamma(q))$. *Let* z *be an integer,* $1 \leq z \leq n$, *and* y *be any positive number. Let* g *be such that* $\gamma(g) = \pi(U, z)$. *Let* $W = (w(1), w(2), \ldots, w(n))$ *be a sequence such that* $w(i) \leq w(\gamma(1))$ *for* $1 \leq i < \gamma(1)$, $w(i) = u(i) - y$ *for* $\gamma(1) \leq i < z$, *and* $w(i) = u(i) + y$ *for* $z \leq i \leq n$. *If* $w(\gamma(1)) \leq w(\gamma(g))$, *let p=0; otherwise, let* p *be the largest index such that* $w(\gamma(p)) > w(\gamma(g))$. *Then, we have* $\Gamma(W) = (\gamma(1), \gamma(2), \ldots, \gamma(p), \gamma(g), \gamma(g+1), \ldots, \gamma(q))$.

3 Algorithm for the RLPW

Our algorithm is a modified version of Myung, Kim, and Tcha's in [5]. Thus, we begin by reviewing their algorithm. Assume that the demands in K are pre-sorted as follows: if $o(k_1) < o(k_2)$, then $k_1 < k_2$, and if $(o(k_1) = o(k_2)$ and $d(k_1) > d(k_2))$, then $k_1 < k_2$. Initially, set $X = (x(1), x(2), \ldots, x(|K|)) = (r(1), r(2), \ldots, r(|K|))$, which indicates that at the beginning all demands are routed in the clockwise direction. Then, for each $k \in K$, the algorithm tries to reduce the maximum load by rerouting all or part of k in the counterclockwise direction. To be more precise, if $\max\{g(X, e_i) | e_i \in E_k^+\} > \max\{g(X, e_i) | e_i \in E_k^-\}$, the algorithm reroutes k until either all the demand is routed in the counterclockwise direction or the resulting X satisfies $\max\{g(X, e_i) | e_i \in E_k^+\} = \max\{g(X, e_i) | e_i \in E_k^-\}$. The algorithm is formally expressed as follows.

Algorithm 1. RLPW-1
Input: an integer n and a set K of demands
Output: a routing $X \in \mathcal{X}$ that minimizes $\max_{1 \leq i \leq n} g(X, e_i)$
begin
1. $X \leftarrow (r(1), r(2), \ldots, r(|K|))$
2. $F \leftarrow (f(1), f(2), \ldots, f(n))$, where $f(i) = g(X, e_i)$
3. **for** $k \leftarrow 1$ **to** $|K|$ **do**
4. **begin**
5. $m(E_k^+) \leftarrow \max\{f(i) | e_i \in E_k^+\}$
6. $m(E_k^-) \leftarrow \max\{f(i) | e_i \in E_k^-\}$
7. **if** $m(E_k^+) > m(E_k^-)$ **then** $y_k \leftarrow \min\{(m(E_k^+) - m(E_k^-))/2, r(k)\}$
8. **else** $y_k \leftarrow 0$
9. $x(k) \leftarrow r(k) - y_k$ /* Reroute y_k units in counterclockwise direction. */
10. Update F by adding y_k to each $f(i)$ with $e_i \in E_k^-$ and subtracting y_k from each $f(i)$ with $e_i \in E_k^+$
11. **end**
12. **return** (X)
end

The bottleneck of Algorithm 1 is the computation of $m(E_k^+)$ and $m(E_k^-)$ for each $k \in K$. In order to obtain a linear time solution, some properties of Algorithm 1 are discussed in the following. Let $X_0 = (r(1), r(2), \ldots, r(|K|))$

and X_k be the X obtained after the rerouting step is performed for $k \in K$. For $0 \leq k \leq |K|$, let $F_k = (f_k(1), f_k(2), \ldots, f_k(n))$, where $f_k(i) = g(X_k, e_i)$. According to the execution of Algorithm 1, once an edge becomes a maximum load edge at some iteration, it remains as such in the remaining iterations. Let $M_k = \{e_i | f_k(i) = max(F_k), 1 \leq i \leq n\}$, which is the set of the maximum load edges with respect to X_k. We have the following.

Lemma 4. *[5] For each $k \in K$, $M_{k-1} \subseteq M_k$.*

Since $m(E_k^+) > m(E_k^-)$ if and only if $m(E_k^+) = max(F_{k-1})$ and $m(E_k^-) \neq max(F_{k-1})$, we have the following lemma.

Lemma 5. *[5] For each $k \in K$, $y_k > 0$ if and only if $E_k^+ \supseteq M_{k-1}$.*

Consider the computation of $m(E_k^+)$ in Algorithm 1. If $m(E_k^+) = max(F_{k-1})$, y_k is computed as $\min\{(max(F_{k-1}) - m(E_k^-))/2, r(k)\}$. Assume that $m(E_k^+) \neq max(F_{k-1})$. In this case, we must have $m(E_k^-) = max(F_{k-1})$. Thus, $m(E_k^+) < m(E_k^-)$ and y_k should be computed as 0, which is irrelevant to the value of $m(E_k^+)$. Since $m(E_k^-) = max(F_{k-1})$, in this case, we can also compute y_k as $\min\{(max(F_{k-1}) - m(E_k^-))/2, r(k)\}$. Therefore, to determine y_k, it is not necessary for us to compute $m(E_k^+)$. What we need is the value of $max(F_{k-1})$. The value of $max(F_0)$ can be computed in $O(n)$ time. According to Lemma 4 and Line 10 of Algorithm 1, after y_k has been determined we can compute $max(F_k)$ as $max(F_{k-1}) - y_k$.

Next, consider the computation of $m(E_k^-)$. In order to compute all $m(E_k^-)$ efficiently, we partition E_k^- into two subsets A_k and B_k, where $A_k = \{e_i | 1 \leq i < o(k)\}$ and $B_k = \{e_i | d(k) \leq i \leq n\}$. For each $k \in K$, we define $m(A_k) = \max\{f_{k-1}(i) | e_i \in A_k\}$ and $m(B_k) = \max\{f_{k-1}(i) | e_i \in B_k\}$. Then, $m(E_k^-) = \max\{m(A_k), m(B_k)\}$. We have the following.

Lemma 6. *Let $k \in K$. If there is an iteration $i < k$ such that $y_i > 0$ and $o(k) > d(i)$, then $y_j = 0$ for all $j \geq k$.*

Proof. Assume that there exists a such i. Since $y_i > 0$, by Lemma 5 we have $E_i^+ \supseteq M_{i-1}$. Consider a fixed $j \geq k$. Since $o(j) \geq o(k) > d(i)$, E_j^+ cannot include M_{i-1}. Furthermore, since by Lemma 4 $M_{j-1} \supseteq M_{i-1}$, E_j^+ cannot include M_{j-1}. Consequently, by Lemma 5, we have $y_j = 0$. Therefore, the lemma holds. □

According to Lemma 6, we may maintain in Algorithm 1 a variable d_{min} to record the current smallest $d(i)$ with $y_i > 0$. Then, at each iteration $k \in K$, we check whether $o(k) > d_{min}$ and once the condition is true, we skip the rerouting for all $j \geq k$. Based upon the above discussion, we present a modified version of Algorithm 1 as follows.

Algorithm 2. RLPW-2
Input: an integer n and a set K of demands
Output: a routing $X \in \mathcal{X}$ that minimizes $\max_{1 \leq i \leq n} g(X, e_i)$
begin
1. $X \leftarrow (r(1), r(2), ..., r(|K|))$

2. $F_0 \leftarrow (f_0(1), f_0(2), ..., f_0(n))$, where $f_0(i) = g(X, e_i)$
3. $max(F_0) \leftarrow \max\{f_0(i)|e_i \in E\}$
4. $d_{min} \leftarrow \infty$
5. **for** $k \leftarrow 1$ **to** $|K|$ **do**
6. **begin**
7. **if** $o(k) > d_{min}$ **then return** (X)
8. $m(A_k) \leftarrow \max\{f_{k-1}(i)|e_i \in A_k\}$
9. $m(B_k) \leftarrow \max\{f_{k-1}(i)|e_i \in B_k\}$
10. $y_k \leftarrow \min\{(max(F_{k-1}) - m(A_k))/2, (max(F_{k-1}) - m(B_k))/2, r(k)\}$
11. $x(k) \leftarrow r(k) - y_k$
12. $max(F_k) \leftarrow max(F_{k-1}) - y_k$
13. **if** $y_k > 0$ and $d(k) < d_{min}$ **then** $d_{min} \leftarrow d(k)$
14. **end**
15. **return** (X)

end

In the remainder of this section, we show that Algorithm 2 can be implemented in linear time. The values of $m(A_k)$, $m(B_k)$, and y_k are defined on the values of F_{k-1}. In Line 2, we compute F_0 in $O(|K|)$ time. Before presenting the details, we remark that our implementation does not compute the whole sequences of all F_{k-1}. Instead, we maintain only their information that is necessary for determining $m(A_k)$, $m(B_k)$, and y_k.

First, we describe the determination of $m(A_k)$, which is mainly based upon the following two lemmas.

Lemma 7. *For each $k \in K$, if Algorithm 2 does not terminate at Line 7, then $f_{k-1}(i) = f_0(i) - \sum_{1 \le i \le k-1} y_i$ for $o(k-1) \le i < o(k)$.*

Proof. Let e_i be an edge such that $o(k-1) \le i < o(k)$. We prove this lemma by showing that $e_i \in E_j^+$ for all $j \le k-1$ and $y_j > 0$. Consider a fixed $j \le k-1$ with $y_j > 0$. Since $o(j) \le o(k-1) \le i$, $o(j)$ is on the left side of e_i. Since Algorithm 2 does not terminate at Line 7, we have $i < o(k) \le d_{min} \le d(j)$. Thus, $d(j)$ is on the right side of e_i. Therefore, $e_i \in E_j^+$ and the lemma holds. □

Lemma 8. *For each $k \in K$, if Algorithm 2 does not terminate at Line 7, then $m(A_k) = \max\{m(A_{k-1}) + y_{k-1}, \max\{f_{k-1}(i)|o(k-1) \le i < o(k)\}\}$, where $m(A_0) = 0$, $y_0 = 0$, and $o(0) = 1$.*

Proof. Recall that $m(A_k) = \max\{f_{k-1}(i)|1 \le i < o(k)\}$. For $k = 1$, since $m(A_0) = 0$, $y_0 = 0$, and $o(0) = 1$, the lemma holds trivially. Assume that $k \ge 2$. In the following, we complete the proof by showing that $m(A_{k-1}) + y_{k-1} = \max\{f_{k-1}(i)|1 \le i < o(k-1)\}$. By induction, $m(A_{k-1}) = \max\{f_{k-2}(i)|1 \le i < o(k-1)\}$. According to Line 10 of Algorithm 1, we have $f_{k-1}(i) = f_{k-2}(i) + y_{k-1}$ for $1 \le i < o(k-1)$. Thus, $\max\{f_{k-1}(i)|1 \le i < o(k-1)\} = \max\{f_{k-2}(i) + y_{k-1}|1 \le i < o(k-1)\} = m(A_{k-1}) + y_{k-1}$. Therefore, the lemma holds. □

According to Lemmas 7 and 8, we compute each $m(A_k)$ as follows. During the execution of Algorithm 2, we maintain an additional variable y^* such that at the beginning of each iteration k, the value of y^* is $\sum_{1 \le i \le k-1} y_i$. Then, for each $k \in K$, if Algorithm 2 does not terminate at Line 7, we compute $m(A_k) = \max\{m(A_{k-1}) + y_{k-1}, \max\{f_0(i) - y^* | o(k-1) \le i < o(k)\}\}$ in $O(o(k) - o(k-1))$ time by using $m(A_{k-1})$, y_{k-1}, F_0, and y^*. Since $2|K| \ge n$ and the origins $o(k)$ are non-decreasing integers between 1 and n, the computation for all $m(A_k)$ takes $O(|K|)$ time.

Next, we describe the determination of $m(B_k)$ and y_k, which is the most complicated part of our algorithm. By definition, $m(B_k) = \max\{f_{k-1}(i) | d(k) \le i \le n\} = f_{k-1}(\pi(F_{k-1}, d(k)))$. Thus, maintaining the function π for F_{k-1} is useful for computing $m(B_k)$. Let $\Gamma(F_{k-1}) = (\gamma(1), \gamma(2), \ldots, \gamma(q))$. For each j, $1 \le j \le q$, we have $\pi(F_{k-1}, i) = \gamma(j)$ for every i in the interval $[\gamma(j-1)+1, \gamma(j)]$, where $\gamma(0) = 0$. Thus, we call $[\gamma(j-1)+1, \gamma(j)]$ the *domain-interval* of $\gamma(j)$. Let U_{k-1} be the sequence of domain-intervals of the elements in $\Gamma(F_{k-1})$. The following lemma, which can be obtained from Lemma 3, shows that U_k can be obtained from U_{k-1} by simply merging the domain-intervals of some consecutive elements in $\Gamma(F_{k-1})$.

Lemma 9. *Let $\Gamma(F_{k-1}) = (\gamma(1), \gamma(2), \ldots, \gamma(q))$. Let g be such that $\gamma(g) = \pi(F_{k-1}, d(k))$. If $f_{k-1}(\gamma(1)) - y_k = f_{k-1}(\gamma(g)) + y_k$, let p=0; otherwise, let p be the largest index such that $f_{k-1}(\gamma(p)) - y_k > f_{k-1}(\gamma(g)) + y_k$. Then, we have $\Gamma(F_k) = (\gamma(1), \gamma(2), \ldots, \gamma(p), \gamma(g), \gamma(g+1), \ldots, \gamma(q))$.*

Based upon Lemma 9, we maintain U_{k-1} by using an interval union-find data structure, which is defined as follows. Let I_n be the interval $[1, n]$. Two intervals in I_n are *adjacent* if they can be obtained by splitting an interval. A *partition* of I_n is a sequence of disjoint intervals whose union is I_n. An *interval union-find data structure* is one that initially represents some partition of I_n and supports a sequence of two operations: FIND(i), which returns the representative of the interval containing i, and UNION(i, j), which unites the two adjacent intervals containing i and j, respectively, into one. The representative of an interval may be any integer contained in it. Gabow and Tarjan had the following result.

Lemma 10. *[3] A sequence of m FIND and at most $n-1$ UNION operations on any partition of I_n can be done in $O(n+m)$ time.*

Let $\Gamma(F_{k-1}) = (\gamma(1), \gamma(2), \ldots, \gamma(q))$. For convenience, we let each $\gamma(i)$ be the representative of its domain-interval such that $\pi(F_{k-1}, d(k))$ can be determined by simply performing FIND($d(k)$). By Lemma 9, U_k can be obtained from U_{k-1} by performing a sequence of UNION operations. In order to obtain U_k in such a way, we need the representatives $\gamma(p+1)$, $\ldots$, and $\gamma(g-1)$. Therefore, we maintain an additional linked list L_{k-1} to chain all representatives in U_{k-1} together such that for any given $\gamma(i)$, we can find $\gamma(i-1)$ in $O(1)$ time.

Now, we can determine $\pi(F_{k-1}, d(k))$ efficiently. However, since $m(B_k) = f_{k-1}(\pi(F_{k-1}, d(k)))$, what we really need is the value of $f_{k-1}(\pi(F_{k-1}, d(k)))$. At this writing, the author is not aware of any efficient way to compute the values

for all $k \in K$. Fortunately, in some case, it is not necessary to compute the value. Since $y_k = \min\{(max(F_{k-1}) - m(A_k))/2, (max(F_{k-1}) - m(B_k))/2, r(k)\}$, the value is needed only when $max(F_{k-1}) - m(B_k) < \min\{max(F_{k-1}) - m(A_k), 2r(k)\}$. Therefore, we maintain further information about F_{k-1} such that whether $max\ (F_{k-1}) - m(B_k) < \min\{max(F_{k-1}) - m(A_k), 2r(k)\}$ can be determined and in case it is true, the value of $m(B_k)$ can be computed. Let $\Gamma(F_{k-1}) = (\gamma(1), \gamma(2), \ldots, \gamma(q))$. We associate with each representative $\gamma(i)$ in L_{k-1} a value $\delta(i)$, where $\delta(i)$ is 0 if i=1 and otherwise $\delta(i)$ is the difference $f_{k-1}(\gamma(i-1)) - f_{k-1}(\gamma(i))$. Define $\Delta(F_{k-1})$ to be the sequence $(\delta(1), \delta(2), \ldots, \delta(q))$. Clearly, for any $i < j$, the difference between $f_{k-1}(\gamma(i))$ and $f_{k-1}(\gamma(j))$ is $\sum_{i+1 \le z \le j} \delta(z)$. And, since $f_{k-1}(\gamma(1)) = max(F_{k-1})$, the difference between $max(F_{k-1})$ and $f_{k-1}(\gamma(i))$ is $\sum_{2 \le z \le i} \delta(z)$. The maintainance of $\Delta(F_{k-1})$ can be done easily by using the following lemma.

Lemma 11. *Let* $\Gamma(F_{k-1}) = (\gamma(1), \gamma(2), \ldots, \gamma(q))$ *and* $\Delta(F_{k-1}) = (\delta(1), \delta(2), \ldots, \delta(q))$. *Let* g *and* p *be defined as in Lemma 9. Then, we have* $\Delta(F_k) = (\delta(1), \delta(2), \ldots, \delta(p), \delta', \delta(g+1), \delta(g+2), \ldots, \delta(q))$, *where* $\delta' = \delta(p+1) + \delta(p+2) + \ldots + \delta(g) - 2y_k$.

Proof. By Lemma 9, we have $\Gamma(F_k) = (\gamma(1), \gamma(2), \ldots, \gamma(p), \gamma(g), \gamma(g+1), \ldots, \gamma(q))$. Since $f_k(\gamma(p)) - f_k(\gamma(g)) = (f_{k-1}(\gamma(p)) - y_k) - (f_{k-1}(\gamma(g)) + y_k)$, we have $f_k(\gamma(p)) - f_k(\gamma(g)) = \delta(p+1) + \delta(p+2) + \ldots + \delta(g) - 2y_k$. Clearly, we have $f_k(\gamma(i-1)) - f_k(\gamma(i)) = f_{k-1}(\gamma(i-1)) - f_{k-1}(\gamma(i))$ for both $2 \le i \le p$ and $g < i \le q$. By combining these two statements, we obtain $\Delta(F_k) = (\delta(1), \delta(2), \ldots, \delta(p), \delta(p+1) + \delta(p+2) + \ldots + \delta(g) - 2y_k, \delta(g+1), \delta(g+2), \ldots, \delta(q))$. Thus, the lemma holds. □

Now, we are ready to describe the detailed computation of y_k, which is done by using $m(A_k)$, $max(F_{k-1})$, $r(k)$, U_{k-1}, L_{k-1}, and $\Delta(F_{k-1})$. First, we perform FIND($d(k)$) to get $\gamma(g) = \pi(F_{k-1}, d(k))$. Next, by traveling along the list L_{k-1}, starting at $\gamma(g)$, we compute the largest p such that $\delta(p+1) + \delta(p+2) + \ldots + \delta(g) > \min\{max(F_{k-1}) - m(A_k), 2r(k)\}$. In case $\delta(1) + \delta(2) + \ldots + \delta(g) \le \min\{max(F_{k-1}) - m(A_k), 2r(k)\}$, p is computed as 0. Then, if $p > 0$, we conclude that $max(F_{k-1}) - m(B_k) = max(F_{k-1}) - f_{k-1}(\gamma(g)) > \min\{max(F_{k-1}) - m(A_k), 2r(k)\}$ and thus y_k is computed as $\min\{(max(F_{k-1}) - m(A_k))/2, r(k)\}$; otherwise, we compute $m(B_k) = f_{k-1}(\gamma(g)) = max(F_{k-1}) - (\delta(1) + \delta(2) + \ldots + \delta(g))$ and then compute y_k as $(max(F_{k-1}) - m(B_k))/2$. Since $g - p - 1 = |\Gamma(F_{k-1})| - |\Gamma(F_k)|$, the above computation takes $t_f + O(|\Gamma(F_{k-1})| - |\Gamma(F_k)|)$ time, where t_f is the time for performing a FIND operation. After y_k is computed, we obtain U_k, L_k, and $\Delta(F_k)$ from U_{k-1}, L_{k-1}, and $\Delta(F_{k-1})$ in $O(|\Gamma(F_{k-1})| - |\Gamma(F_k)|) + (|\Gamma(F_{k-1})| - |\Gamma(F_k)|) \times t_u$ time according to Lemmas 9 and 11, where t_u is the time for performing an UNION operation.

Theorem 1. *The RLPW can be solved in* $O(|K| + t_s)$ *time, where* t_s *is the time for sorting* $|K|$ *nodes.*

Proof. We prove this theorem by showing that Algorithm 2 can be implemented in $O(|K|)$ time. Note that we had assumed $2|K| \ge n$. The time for compute X,

F_0, $max(F_0)$, and d_{min} in Lines 1∼4 is $O(|K|)$. Before starting the rerouting, we set $y^* = 0$, $m(A_0) = 0$, $y_0 = 0$, and initialize U_0, L_0, and $\Delta(F_0)$ in $O(n)$ time. Consider the rerouting in Lines 7∼13 for a fixed $k \in K$. In Line 8, by using Lemmas 7 and 8, we compute $m(A_k)$ by using $m(A_{k-1}), y_{k-1}$, F_0 and y^* in $O(o(k) - o(k-1))$ time. In Lines 9 and 10, we compute y_k in $t_f + O(|\Gamma(F_{k-1})| - |\Gamma(F_k)|)$ time by using $m(A_k)$, $max(F_{k-1})$, $r(k)$, U_{k-1}, L_{k-1}, and $\Delta(F_{k-1})$. Lines 7, 11, 12, and 13 take $O(1)$ time. Before starting the next iteration, we add y_k to y^* and obtain U_k, L_k, and $\Delta(F_k)$ from U_{k-1}, L_{k-1}, and $\Delta(F_{k-1})$ in $O(|\Gamma(F_{k-1})| - |\Gamma(F_k)|) + (|\Gamma(F_{k-1})| - |\Gamma(F_k)|) \times t_u$ time. In total, the rerouting time for a fixed $k \in K$ is $t_f + (|\Gamma(F_{k-1})| - |\Gamma(F_k)|) \times t_u + O(o(k) - o(k-1) + |\Gamma(F_{k-1})| - |\Gamma(F_k)|)$.

Since the origins $o(k)$ are non-decreasing and the sizes of $\Gamma(F_{k-1})$ are non-increasing, $\sum_{1 \le i \le |K|} O(o(k) - o(k-1) + |\Gamma(F_{k-1})| - |\Gamma(F_k)|) = O(|K|)$. At most $|K|$ FIND and $n-1$ UNION operations may be performed. Therefore, the overall time complexity of Algorithm 2 is $O(|K| + |K| \times t_f + n \times t_u)$, which is $O(|K|)$ by applying Gabow and Tarjan's result in Lemma 10. Consequently, the theorem holds. □

4 Algorithm for the RLPWI

The algorithm proposed by Myung for the RLPWI in [7] consists of two phases. In the first phase, an optimal solution X for the RLPW is found. Then, if $X \notin \mathcal{X} \bigcap Z^{|K|}$, the second phase is performed, in which demands are rerouted until all $x(k)$ become integers. The bottleneck of Myung's algorithm is the computation of X in the first phase and the computation of all $g(X, e_i)$ in the second phase. By using Theorem 1 and Lemma 1, it is easy to implement Myung's algorithm in $O(|K| + t_s)$ time.

Theorem 2. *The RLPWI can be solved in $O(|K| + t_s)$ time, where t_s is the time for sorting $|K|$ nodes.*

5 Concluding Remarks

In this paper, an $O(|K| + t_s)$-time algorithm was firstly proposed for the RLPW. Then, by applying it to Myung's algorithm in [6], the RLPWI was solved in the same time. The proposed algorithms take linear time when $|K| \ge n^\epsilon$ for some small constant $\epsilon > 0$. They improved the previous upper bounds from $O(n|K|)$ for both problems.

Myung, Kim, and Tcha's algorithm for the RLPW in [5] motivated studies on the following interesting data structure. Let $X = (x_1, x_2, \ldots, x_n)$ be a sequence of n values. A *range increase-decrease-maximum data structure* is one that initially represents X and supports a sequence of three operations: INCREASE(i, j, y), which adds y to every element in $(x_i, x_{i+1}, \ldots, x_j)$, DECREASE($i$, j, y), which subtracts y from every element in $(x_i, x_{i+1}, \ldots, x_j)$, and MAXIMUM($i$, j),

which returns the maximum in $(x_i, x_{i+1}, \ldots, x_j)$. By using the well-known segment trees, it is not difficult to implement a data structure to support each of the three operations in $O(\log n)$ time. To design a more efficient implementation of such data structure is also worth of further study.

References

1. M. D. Amico, M. Labbe, and F. Maffioli, "Exact solution of the SONET ring loading problem," *Operations Research Letters*, vol. 25, pp. 119–129, 1999.
2. S. Cosares and I. Saniee, "An optimal problem related to balancing loads on SONET rings," *Telecommunication Systems*, vol. 3, pp. 165–181, 1994.
3. H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union," *Journal of Computer and System Sciences*, vol. 30, pp. 209–221, 1985.
4. C. Y. Lee and S. G. Chang, "Balancing loads on SONET rings with integer demand splitting," *Computers Operations Research*, vol. 24, pp. 221–229, 1997.
5. Y.-S. Myung, H.-G. Kim, and D.-W. Tcha, "Optimal load balancing on SONET bidirectional rings," *Operations Research*, vol. 45, pp. 148–152, 1997.
6. Y.-S. Myung, "An efficient algorithm for the ring loading problem with integer demand splitting," *SIAM Journal on Discrete Mathematics*, vol. 14, no. 3, pp. 291–298, 2001.
7. A. Schrijver, P. Seymour, and P. Winkler, "The ring loading problem," *SIAM Journal on Discrete Mathematics*, vol. 11, pp. 1–14, 1998.
8. R. Vachani, A. Shulman, and P. Kubat, "Multi-commodity flows in ring networks," *INFORMS Journal on Computing*, vol. 8, pp. 235–242, 1996.