Journal Article

# Call control in rings

**Author(s):**
Adamy, Udo; Ambruehl, Christoph; Anand, R. Sai; Erlebach, Thomas

**Publication Date:**
2007-03

**Permanent Link:**
https://doi.org/10.3929/ethz-b-000068034 →

**Originally published in:**
Algorithmica 47(3), http://doi.org/10.1007/s00453-006-0187-4 →

**Rights / License:**
In Copyright - Non-Commercial Use Permitted →

ETH Library

# Call Control in Rings[1]

Udo Adamy,[2] Christoph Ambühl,[3] R. Sai Anand,[4] and Thomas Erlebach[5]

**Abstract.** The call control problem is an important optimization problem encountered in the design and operation of communication networks. The goal of the call control problem in rings is to compute, for a given ring network with edge capacities and a set of paths in the ring, a maximum cardinality subset of the paths such that no edge capacity is violated. We give a polynomial-time algorithm to solve the problem optimally. The algorithm is based on a decision procedure that checks whether a solution with at least $k$ paths exists, which is in turn implemented by an iterative greedy approach operating in rounds. We show that the algorithm can be implemented efficiently and, as a by-product, obtain a linear-time algorithm to solve the problem in chains optimally. For the weighted version of call control in rings, where each path is associated with a weight and the goal is to maximize the total weight of the paths in the solution, we present a simple 2-approximation algorithm and a polynomial-time approximation scheme. While the complexity of the weighted version remains open, we show that it is at least as hard as the bipartite exact matching problem, which has not been resolved to be in $P$ or $NP$-hard. This latter result follows from recent work by Hochbaum and Levin.

**1. Introduction.** Due to the ever-increasing importance of communication networks for our society and economy, optimization problems concerning the efficient operation of such networks are receiving considerable attention in the research community. Many of these problems can be modeled as graph problems or path problems in graphs. A prominent example is the problem of *call admission control* (or simply *call control*), where the task is to determine which of the requests in a given set of connection requests (calls) to accept or reject so as to optimize some objective, e.g., the number of accepted requests or the total weight of the accepted requests.

The ring topology is a fundamental network topology that is frequently encountered in practice. In this paper we consider the off-line call control problem in ring networks

with arbitrary edge capacities, assuming that each request asks for one unit of bandwidth on all edges of its predetermined path. For the objective of maximizing the number of accepted paths, we present an efficient polynomial-time algorithm that solves the problem optimally. As a by-product, we obtain a linear-time algorithm for chains. For the case that the paths have weights (profits) and the goal is to maximize the total weight of the accepted paths, we present a simple 2-approximation algorithm and a polynomial-time approximation scheme (PTAS) for rings. While it remains an open problem whether the weighted case can be solved optimally in polynomial time, we provide an indication of its difficulty: based on recent results by Hochbaum and Levin [11], we show that a polynomial-time algorithm for weighted call control in rings would also solve the bipartite exact matching problem. The latter problem has a randomized polynomial-time algorithm [15], but it is a long-standing open problem whether it can be solved in polynomial time by a deterministic algorithm.

Call control in rings is closely related to cyclical scheduling. We discuss how our optimal algorithm for call control in rings can be used to solve such scheduling problems as well.

1.1. *Problem Definition and Preliminaries*.    The CALLCONTROL problem considered in this paper is defined as follows. An instance of the problem is given by an undirected graph $(V, E)$ with edge capacities $c: E \to \mathbf{N}$ and a set $P$ of $m$ paths in $(V, E)$. In general, the set of paths $P$ is a multiset, i.e., it may contain several instances of the same path. The paths represent connection requests whose acceptance requires the reservation of one unit of bandwidth on all edges of the path. A feasible solution is a set $Q \subseteq P$ such that for every edge $e \in E$, the number of paths in $Q$ that contain $e$ is at most $c(e)$. Such a set of paths is called a *feasible set* and the paths in it are called *accepted*. The objective is to maximize the number of accepted paths.

We also consider the weighted version of the problem and refer to it as WEIGHTED-CALLCONTROL. Here, an instance of the problem contains additionally a weight function on the paths, $w: P \to \mathbf{N}$, and the objective is to maximize the sum of the weights of the accepted paths. Note that CALLCONTROL corresponds to the special case of WEIGHTED-CALLCONTROL in which all weights are equal.

In this paper we deal with CALLCONTROL and WEIGHTEDCALLCONTROL in networks with ring or chain topology. A *ring* with $n$ nodes is an undirected graph $(V, E)$ that is a cycle on the nodes $V = \{0, \ldots, n-1\}$. We imagine the cycle drawn in the plane with the nodes labeled clockwise. The edge $e_i \in E$, $0 \leq i < n$, connects the two neighboring nodes $i$ and $(i+1) \bmod n$ and has a non-negative integer capacity $c(e_i)$. A *chain* is an undirected graph that consists of a single path. In instances of CALLCONTROL in rings or chains, we assume that each path is specified by its endnodes (in rings, the counterclockwise and clockwise endnode).

The problem of CALLCONTROL in ring networks as defined above applies to various types of existing communication networks with ring topology. For example, the problem applies to ring networks that support bandwidth reservation (e.g., ATM networks) and in which the route taken by a request is determined by some other mechanism and cannot be modified by the call control algorithm. Furthermore, it applies to bidirectional self-healing rings with full protection. In such rings, one direction of the ring (say,

clockwise) is used to route all accepted requests during normal operation, and the other direction is used only in case of a link failure in order to reroute the active connections that are affected. In all-optical WDM ring networks with $w$ wavelengths that have a wavelength converter in at least one node, any set of lightpaths with maximum link load $w$ can be established simultaneously [18]. Thus, call admission control in such networks can also be modeled as CALLCONTROL with all edge capacities equal to $w$. Another application is in ring networks with several parallel links connecting consecutive nodes.

It should be noted that problems related to call control are often encountered in an on-line setting, where the requests are presented to the algorithm one by one and the algorithm must accept or reject each request without knowledge of future requests. However, we think that it is meaningful to study the off-line version as well for several reasons. First, an off-line call control algorithm is needed in the network *design* phase, when a candidate topology with link capacities is considered and one wants to know how many of the forecasted traffic requirements can be satisfied by the network. Second, an off-line call control algorithm is useful in a scenario that supports advance reservation of connections, because then it is possible to collect a number of reservation requests before the admission control is carried out for a whole batch of requests. Finally, an optimal off-line call control algorithm is helpful as a benchmark for the evaluation of other off-line or on-line call control strategies.

1.2. *Applications in Cyclical Scheduling.* Call control in rings is closely related to cyclical (or periodic) scheduling. First, consider the following setting. Without loss of generality, assume a time period of one day. There are $k$ machines and a set of tasks with fixed start and end times. (For example, there could be a task from 10 a.m. to 5 p.m. and another task from 3 p.m. to 2 a.m. on the following day.) Each task must be accepted or rejected. If it is accepted, it must be executed every day from its start time to its end time on one of the $k$ machines, and each machine can execute only one task at a time. The goal is to select as many tasks as possible while ensuring that at most $k$ of the selected tasks are to be executed simultaneously at any point in time. By taking the start times and end times of all given tasks as nodes in a ring, we can view the tasks as calls and compute an optimal selection of accepted tasks by solving the corresponding CALLCONTROL problem with all edge capacities set to $k$. Here, we assume that a task can be executed on different machines on different days or that a task can be moved from one machine to another at any time. Even if the number of available machines changes throughout the day (and the changes are the same every day), the problem can still be handled as a CALLCONTROL problem with arbitrary edge capacities.

Hochbaum and Levin [11] discuss a different setting, in which there are requirements for a specific number of workers in each time slot throughout the time period. The goal is to satisfy these requirements using work shifts of minimum total cost, where each shift covers a collection of consecutive time slots. For example, problems of this type arise in the management of personnel in hospitals. They can be modeled as a bounded multicover problem, which has the following representation as an integer linear

program (ILP):

$$\text{(MCB)} \quad \min \sum_{j=1}^{m} c_j x_j$$
$$\text{s.t.} \sum_{j=1}^{m} a_{ij} x_j \geq b_i, \qquad i = 1, \ldots, n,$$
$$0 \leq x_j \leq u_j, \qquad j = 1, \ldots, m,$$
$$x_j \text{ integer.}$$

Here, $A = (a_{ij})$ is an $n \times m$ matrix with entries in $\{0, 1\}$. Each column of $A$ corresponds to one possible shift, and the cost associated with shift $j$ is $c_j$. The minimum number of workers required in the $i$th time slot is given by $b_i$, $1 \leq i \leq n$. The variable $x_j$ represents the number of workers that are assigned to shift $j$, and $u_j$ is an upper bound for this number. Since each shift consists of consecutive time slots, the matrix $A$ has the circular-ones property in its columns, i.e., the ones in each column form a consecutive interval if the first row of the matrix is considered to be the successor of the last row.

For the case where all $b_i$ are equal to 1, polynomial-time algorithms for MCB were presented by Atallah et al. [3] and by Hochbaum and Tucker [12]. If all $b_i$ are equal but possibly larger than 1, the unbounded version of MCB (with $u_j = \infty$ for all $j$) is shown to be polynomial by Hochbaum and Levin [11]. For the general MCB, they present a 2-approximation algorithm. Our polynomial-time algorithm for CALLCONTROL can be used to solve MCB optimally in polynomial time provided that $c_j = 1$ for all $j$ and the $u_j$'s are polynomially bounded in $n$ and $m$. First, we simply replace each variable $x_j$ by $u_j$ copies of itself and assign each copy an upper bound of 1. Since the $u_j$'s are polynomially bounded, the resulting instance has polynomial size. Then we substitute $y_j = 1 - x_j$ for each of the resulting variables. In this way we obtain the following problem:

$$\min \ m - \sum_{j=1}^{m} y_j$$
$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \leq -b_i + \sum_{j=1}^{m} a_{ij}, \qquad i = 1, \ldots, n,$$
$$0 \leq y_j \leq 1, \qquad j = 1, \ldots, m,$$
$$y_j \text{ integer.}$$

Note that the right-hand sides of the form $-b_i + \sum_{j=1}^{m} a_{ij}$ (which can be assumed to be non-negative, since otherwise the MCB problem has no feasible solution) are fixed values that depend only on $b$ and $A$, and that the objective $\min m - \sum_{j=1}^{m} y_j$ is equivalent to $\max \sum_{j=1}^{m} y_j$ with respect to the values of the variables in the optimal solution. Thus

the problem is of the following form:

$$\max \sum_{j=1}^{m} y_j$$
$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \le c_i, \qquad i = 1, \ldots, n,$$
$$0 \le y_j \le 1, \qquad j = 1, \ldots, m,$$
$$y_j \text{ integer.}$$

This is in fact the natural ILP formulation of the CALLCONTROL problem in rings (where the ones in the $j$th column of $A$ correspond to the edges used by the $j$th path, and $c_i$ represents the capacity of the $i$th edge). Hence, an optimal solution can be computed using our algorithm, and we obtain the claimed polynomial-time optimal algorithm for MCB with unit costs and polynomially bounded $u_j$-values.

Note that the above reduction from MCB to CALLCONTROL in rings works also in the weighted case (i.e., an MCB instance with arbitrary costs is reduced to an instance of WEIGHTEDCALLCONTROL). However, we do not have a polynomial-time optimal algorithm for WEIGHTEDCALLCONTROL in rings, and our approximation algorithms from Sections 4.2 and 4.3 do not yield approximation algorithms for MCB, since the objective functions are not comparable (MCB corresponds to the objective of minimizing the weight of the rejected calls, whereas in WEIGHTEDCALLCONTROL the weight of the accepted calls is to be maximized).

1.3. *Further Related Work.* As paths in a ring network can be viewed as arcs on a circle, path problems in rings are closely related to *circular-arc graphs*. A graph is a circular-arc graph if its vertices can be represented by arcs on a circle such that two vertices are joined by an edge if and only if the corresponding arcs intersect [10]. For a given circular-arc graph, a maximum clique or a maximum independent set can be computed in polynomial time [10]. Coloring a circular-arc graph with the minimum number of colors is *NP*-hard [9]. A coloring with at most $1.5\,\omega$ colors always exists and can be computed efficiently [13], where $\omega$ is the size of a maximum clique in the graph. Concerning our WEIGHTEDCALLCONTROL problem, we note that the special case where all edges have capacity 1 is equivalent to the maximum weight independent set problem in circular-arc graphs. We are interested in the case of arbitrary edge capacities, which has not been studied previously.

Many authors have investigated call control problems for various network topologies in the off-line and on-line setting. For topologies containing cycles, an important distinction for call control is whether the paths are specified as part of the input (like we assume in this paper) or can be determined by the algorithm. In the latter case only the endpoints are specified in the input, and we refer to the problem as CALLCONTROL-ANDROUTING. The special case of CALLCONTROLANDROUTING where all edges have capacity 1 is called the *maximum edge-disjoint paths* problem (MEDP). We refer to [5, Chapter 13], [14], and [16] for surveys on on-line algorithms for call control problems and mention only some of the known results here.

In [2] a polynomial-time algorithm for CALLCONTROLANDROUTING in rings is presented that always computes a feasible solution with at least $OPT - 3$ accepted calls, where $OPT$ is the number of calls in the optimal solution. To the best of our knowledge, it is not known whether CALLCONTROLANDROUTING in rings is *NP*-hard.

For chains, the off-line version of CALLCONTROL is closely related to the maximum $k$-colorable induced subgraph problem for interval graphs. The latter problem can be solved optimally in linear time by a clever implementation of a greedy algorithm provided that a sorted list of interval endpoints is given [6]. This immediately gives a linear-time algorithm for CALLCONTROL in chains where all edges have the same capacity. It is not difficult to adapt the approach to chains with arbitrary capacities incurring an increase in running-time. As a by-product of our algorithm for rings, we will obtain even a linear-time algorithm for CALLCONTROL in chains with arbitrary capacities.

The on-line version of CALLCONTROL in chains with unit edge capacities was studied for the case with preemption (where interrupting and discarding a call that was accepted earlier is allowed) in [8], where competitive ratio $O(\log n)$ is achieved for a chain with $n$ nodes by a deterministic algorithm. A randomized preemptive $O(1)$-competitive algorithm for CALLCONTROL in chains where all edges have the same capacity is given in [1]. It can be adapted to ring networks with equal edge capacities.

In [4] the preemptive on-line version of CALLCONTROL is studied with the number of *rejected* calls as the objective function. They obtain competitive ratio 2 for chains with arbitrary capacities, 2 for arbitrary graphs with unit capacities, and $O(\sqrt{m})$ for arbitrary graphs with $m$ edges and arbitrary capacities. For the off-line version, they give an $O(\log m)$-approximation algorithm for arbitrary graphs and arbitrary capacities.

1.4. *Organization.*    The remainder of the paper is organized as follows. In Section 2 we present our polynomial-time optimal algorithm for CALLCONTROL in rings and analyze its correctness and running-time. As a subroutine, we employ a linear-time implementation of the greedy algorithm that is optimal for CALLCONTROL in chains, and this by-product of our research is described in Section 3. Section 4 presents a 2-approximation algorithm and a PTAS for the WEIGHTEDCALLCONTROL problem in rings and discusses its connection with the bipartite exact matching problem. Finally, we conclude with open problems in Section 5.

**2. An Optimal Algorithm for Call Control in Rings.**    We start with some additional preliminaries. Let $P = \{p_1, \ldots, p_m\}$ denote the given set of $m$ paths, each connecting two nodes in the ring network $(V, E)$ with $n = |V|$ nodes. Every $p_i \in P$ is an ordered pair of nodes $p_i = (s_i, t_i)$ with $s_i \neq t_i$. The path $p_i$ contains all edges from the *source node* $s_i$ to the *target node* $t_i$ in the clockwise direction. The nodes that lie between $s_i$ and $t_i$ in the clockwise direction (excluding $s_i$ and $t_i$) are called *internal nodes* of $p_i$. We can assume without loss of generality that $n \leq 2m$, since every node in the ring that is not an endpoint of a path can be removed. Figure 1 shows a set of seven paths in a ring network on eight nodes. For example, the path $p_1$ is given by the pair $(0, 3)$ of nodes and contains the edges $e_0$, $e_1$, and $e_2$.

For a subset $Q \subseteq P$, the *ringload* $L(Q, e_i)$ of an edge $e_i$ with respect to $Q$ is the number of paths in $Q$ that contain the edge $e_i$, i.e., $L(Q, e_i) := |\{p \in Q : e_i \in p\}|$. A
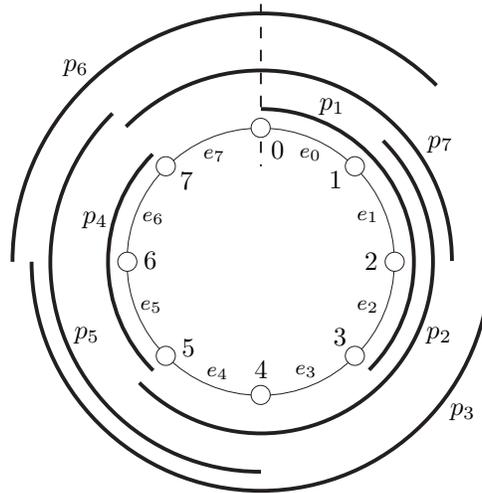
**Fig. 1.** A set of paths in a ring network with eight nodes.

subset $Q \subseteq P$ is called *feasible* if the ringload of any edge does not exceed its capacity, i.e., $L(Q, e_i) \leq c(e_i)$ for all $e_i \in E$.

By opening the ring at node 0, we partition the set $P$ of paths into two disjoint subsets $P_1$ and $P_2$, where $P_1$ is the set of paths that do not have node 0 as an internal node and $P_2$ are the remaining paths, i.e., the paths passing through node 0. Each path in $P_2$ consists of two pieces: the *head* of the path extends from its source node to node 0, and the *tail* spans from node 0 to its target node. To simplify the explanation, we introduce an additional node $n$ and identify it with node 0. From now on, the paths with target node 0 are treated as if they ended at node $n$. With this twist in notation we get a simple characterization for the sets $P_1$ and $P_2$. The set $P_1$ consists of all paths $p_i \in P$ with $s_i < t_i$, whereas all paths $p_i \in P$ obeying $s_i > t_i$ constitute the set $P_2$. Note that $P_1$ and $P_2$ are disjoint and their union $P_1 \cup P_2$ equals $P$. In the example of Figure 1 the set $P_1$ consists of the paths $p_1, \ldots, p_5$, and the set $P_2$ is composed of the paths $p_6$ and $p_7$.

We define a linear ordering on the paths in $P$ as follows. All paths in $P_1$ are strictly less than all paths in $P_2$. Within both subsets we order the paths by increasing target nodes, resolving ties arbitrarily. We call this ordering *greedy*. In the above example the paths $p_1, \ldots, p_7$ are numbered according to that greedy order.

Instead of the ring network the algorithm considers a chain of $2n$ edges consisting of two copies of the ring glued together as shown in Figure 2. The chain begins with the first copy of $e_0$ and ends with the second copy of $e_{n-1}$. The tails of the $P_2$-paths extend in the second copy of the ring, while the $P_1$-paths and the heads of the $P_2$-paths are situated in the first copy. Note that the greedy order of the paths corresponds to an ordering by non-decreasing right endpoints in this chain.

For a given set $Q$ of paths, we define $L_1(Q, e_i)$ and $L_2(Q, e_i)$ to be the load of the paths in $Q$ on the first copy of $e_i$ and their load on the second copy of $e_i$, respectively. Thus, the paths in $P_1$ and the heads of the paths in $P_2$ contribute to the load values $L_1(P, e_i)$ of the first copy of the ring. The tails of the $P_2$-paths determine the load values
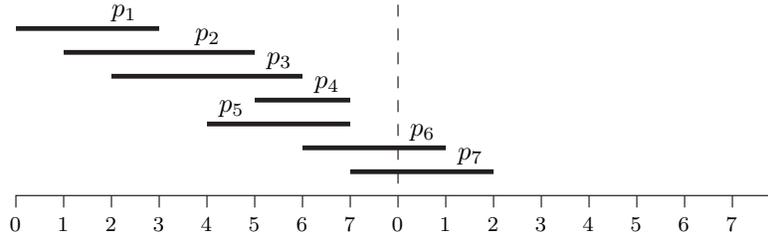
**Fig. 2.** The same set of paths in the chain with 16 nodes.

$L_2(P, e_i)$. The ringload $L$ is simply the sum of $L_1$ and $L_2$. We use this definition of $L_2$ to introduce the central notion of profiles.

DEFINITION 1 (Profiles).   Let $Q$ be a set of paths. The profile $\pi$ of $Q$ is the non-increasing sequence of $n$ load values $L_2$ for the edges $e_0, \ldots, e_{n-1}$ in the second copy of the ring,

$$\pi_Q := L_2(Q, e_0) \cdots L_2(Q, e_{n-1}).$$

With $\pi_Q(e_i)$ we denote the profile values $L_2(Q, e_i)$ for all edges $e_i \in E$. The empty profile is zero everywhere. For profiles $\pi$ and $\pi'$, we write $\pi \leq \pi'$ if $\pi(e_i) \leq \pi'(e_i)$ for all edges $e_i \in E$.

In the instance shown in Figure 2, only the paths $p_6$ and $p_7$ potentially contribute to profile values, because they extend into the second copy of the ring. For example the profile of all paths in $P$ is given by the sequence $\pi_P = 21000000$.

A set $Q$ of paths is called *chain-feasible* if it does not exceed the capacity of any edge in the constructed chain of length $2n$. In other words, $Q$ is *chain-feasible* if it does not exceed the capacities in both copies of the ring, i.e., $L_1(Q, e_i) \leq c(e_i)$ and $L_2(Q, e_i) \leq c(e_i)$ hold for all $e_i \in E$. It is called *chain-feasible for* (*starting*) *profile* $\pi$ if it is chain-feasible and in the first copy of the ring the stronger inequalities $L_1(Q, e_i) + \pi(e_i) \leq c(e_i)$ hold for all $e_i \in E$. Observe that a set $Q$ of paths is feasible (in the ring) if and only if it is chain-feasible for starting profile $\pi_Q$, which is the profile generated by the set $Q$ itself.

2.1. *The Algorithm.*   The goal of the algorithm is to find a feasible subset of paths in $P$ of maximum size. The algorithm builds a chain of $2n$ edges consisting of two copies of the ring glued together. It sorts the paths in $P$ according to the greedy order. The heart of our algorithm is a decision procedure that, given a positive integer parameter $k$, decides whether there exists a feasible solution $Q \subseteq P$ of size $k$ or not. Clearly, the maximum $k$ can be found by several calls to this procedure. The decision procedure makes heavy use of the *greedy algorithm*, which processes the paths one by one in greedy order. If adding the current path does not exceed any capacity constraint on its edges, the path is accepted and the edge capacities are reduced accordingly; otherwise it is rejected.

We are now ready to describe the decision procedure. We start with the empty profile. The decision procedure works in rounds. In each round it computes a (not necessarily feasible) greedy solution of $k$ paths for a given profile as follows. It initializes both

copies of the ring with the edge capacities $c(e_i)$ and subtracts the profile values from the initial capacities of the edges in the first copy, since these capacities are occupied by the profile. Then it starts to place $k$ paths using the greedy algorithm. If the procedure runs out of paths before it can select $k$ of them, there is no feasible solution of size $k$. It answers "no" and stops. Otherwise, let $Q_i$ denote the candidate set of $k$ chosen paths in round $i$. By construction, the set $Q_i$ is chain-feasible for the given starting profile, but not necessarily feasible in the ring, since the tails of the chosen $P_2$-paths together with the selected paths in $P_1$ and the heads of the chosen $P_2$-paths may violate some capacity constraints.

At the end of round $i$, the procedure compares the profile of $Q_i$ with the profile of the previous round. If they are equal, the paths in $Q_i$ form a feasible solution of size $k$. The procedure outputs $Q_i$, answers "yes", and stops. Otherwise, the procedure uses the profile of $Q_i$ as the starting point for round $i + 1$. As we will prove later, the profiles of such a greedily chosen $Q_i$ serve as a lower bound for any feasible solution in the sense that there exists no feasible solution with a smaller profile.

In Figure 3 we illustrate the decision procedure for the ring network and paths shown in Figure 1. Let the capacities be $c(e_i) = 2$ for all edges $e_i$, $i = 0, \ldots, 7$. We ask for a feasible solution consisting of $k = 5$ paths. The paths are always processed in greedy order, which can be seen in Figure 2. We start with the empty profile. In the first round the paths $p_1$ and $p_2$ are accepted. The path $p_3$ is rejected, because it violates the capacity constraint of the edge $e_2$ after the paths $p_1$ and $p_2$ have already been accepted. The paths $p_4$ and $p_5$ are both accepted, and the path $p_6$ is rejected, because otherwise the edge $e_6$ would be overloaded. Finally, the path $p_7$ is accepted to form the candidate set $Q_1 = \{p_1, p_2, p_4, p_5, p_7\}$ of five paths shown in Figure 3(a). The profile of $Q_1$ is 1 on the first two edges $e_0$ and $e_1$, and 0 elsewhere. $Q_1$ is not feasible because the load $L(Q_1, e_1) = 3$ exceeds the capacity $c(e_1) = 2$.

The procedure starts a second round, this time with the profile of $Q_1$ as the starting profile. In this round the procedure accepts the paths in $Q_2 = \{p_1, p_3, p_4, p_6, p_7\}$ illustrated in Figure 3(b). The path $p_2$ is rejected this time, because the edge $e_1$ is saturated by the profile of $Q_1$ and the path $p_1$ that is already accepted. The path $p_5$ is rejected because of the capacity constraint of the edge $e_5$. The profile of $Q_2$ is 2 for the edge $e_0$, 1 for the edge $e_1$, and 0 elsewhere. Like $Q_1$, the candidate set $Q_2$ is not feasible. Its load on the edge $e_0$ is 3, violating the capacity constraint for edge $e_0$, which has capacity $c(e_0) = 2$.

In the third round the procedure starts with the profile of $Q_2$, and accepts the paths in $Q_3 = \{p_2, p_3, p_4, p_6, p_7\}$ depicted in Figure 3(c). Their profile is again 2 for the edge $e_0$, 1 for the edge $e_1$, and 0 elsewhere. Since this resulting profile $\pi_{Q_3}$ is equal to the starting profile $\pi_{Q_2}$, the set $Q_3$ is a feasible solution for the ring of size 5. The procedure outputs $Q_3$, and stops.

2.2. *Correctness of the Algorithm.*   The decision procedure will generate a sequence of profiles and chain-feasible solutions

$$\pi_0 \quad Q_1 \quad \pi_1 \quad Q_2 \quad \pi_2 \quad \ldots,$$

where $\pi_0$ is the empty profile we start with, and $Q_i$ denotes the chain-feasible solution computed in round $i$. We set $\pi_i := \pi_{Q_i}$.
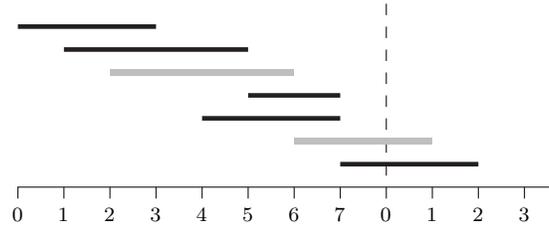
(a) The candidate set $Q_1$ of the first round.
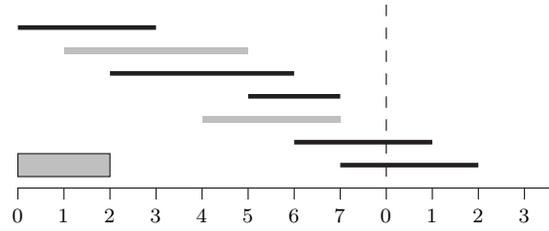


(b) The candidate set $Q_2$ of the second round. The profile $\pi_{Q_1}$ is shown in grey.
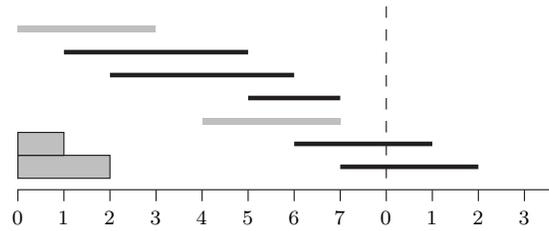


(c) The set $Q_3$ forms a feasible solution.

**Fig. 3.** The decision procedure in rounds. Accepted paths are plotted black, whereas rejected paths are shown in light grey. In this example, $k = 5$ and all edges have capacity equal to 2.

We represent a chain-feasible solution $A$ by the indices of the chosen paths in greedy order. A chain-feasible set $A$ of $k$ paths corresponds to a $k$-vector $A = (a_1, a_2, \ldots, a_k)$, where $a_i$ is the index of the $i$th path in $A$ according to the greedy order. If $A$ and $B$ are two chain-feasible solutions, we write $A \leq B$ if $a_i \leq b_i$ for all $1 \leq i \leq k$.

Note that $A \leq B$ implies $\pi_A \leq \pi_B$. This can be seen by comparing the $i$th path in $A$ with the $i$th path in $B$: since their indices $a_i$ and $b_i$ satisfy the condition $a_i \leq b_i$ for all $i$, the paths in $A$ contribute no more to the profile values $\pi_A(e_j)$ than the paths in $B$ add to their respective profile values $\pi_B(e_j)$ for all edges $e_j$. Thus, $\pi_A \leq \pi_B$.

From $\pi \leq \pi'$ it follows easily that any chain-feasible solution for profile $\pi'$ is also chain-feasible for profile $\pi$. In the following we call a solution $A$ that is chain-feasible for profile $\pi$ *minimal* if for any other solution $B$ that is chain-feasible for $\pi$ and has the same cardinality as $A$, we have $A \leq B$.

LEMMA 1 (Optimality of Greedy Algorithm). *Let $\pi$ be some starting profile. If there exists a solution of size $k$ that is chain-feasible for profile $\pi$, there is also a minimal such solution, and the greedy algorithm computes this minimal solution.*

PROOF.   Let $Q$ be any chain-feasible solution for profile $\pi$ of size $k$. We transform $Q$ step by step into the greedy solution $G$ by replacing paths in $Q$ by paths in $G$ with smaller index. This is done during the execution of the greedy algorithm as it processes the paths in greedy order. We maintain the invariant that $Q$ is always a chain-feasible solution of size $k$ and that $Q$ is equal to $G$ with respect to the paths that have been processed so far.

Initially, the invariant clearly holds. Suppose the invariant holds up to path $p_{i-1}$, and the greedy algorithm processes the path $p_i$.

If adding the path $p_i$ violates some capacity constraint, $p_i$ is not selected by the greedy algorithm. Because of the invariant, the path $p_i$ is not in $Q$ either. Otherwise, the path $p_i$ is chosen by the greedy algorithm. We distinguish two cases:

*Case* 1: $p_i \in Q$.   Since the path $p_i$ is in both $G$ and $Q$, no transformation is needed, and $Q$ remains feasible.

*Case* 2: $p_i \notin Q$.   From the set of paths in $Q$ with indices larger than $i$, we select a path $p_j$ with the smallest source node (starting leftmost). We transform $Q$ by replacing $p_j$ by $p_i$. Since $j > i$, the index $j$ in $Q$ is reduced to $i$. We have to check the invariant. If the path $p_i$ is contained in $p_j$, the invariant clearly holds, since replacing $p_j$ by $p_i$ does not affect feasibility. Otherwise, consider the remaining capacities. The edges to the left of the path $p_j$ do not matter, because $p_j$ has the smallest source node among all paths in $Q$ greater than $p_i$. On the edges in the intersection of the paths $p_i$ and $p_j$, taking either path $p_i$ or $p_j$ does not affect the capacities. Finally, we even gain one unit of capacity on all edges between the target node of the path $p_i$ and the target node of the path $p_j$, since $i < j$. Altogether, $Q$ is again feasible. The invariant holds.

At the end of the greedy algorithm, $Q$ equals $G$. During the transformation we always replaced paths $p_j \in Q$ by paths $p_i \in G$ with $i < j$. This implies that $G$ is less than or equal to the initial chain-feasible solution $Q$, i.e., $G \leq Q$.                                                    □

LEMMA 2.   *The sequence of profiles generated by the decision procedure is monotonically increasing, i.e., we have $\pi_i \leq \pi_{i+1}$ for all $i$.*

PROOF (by induction).   For $i = 0$, the claim holds, since the profile $\pi_0$ is the empty profile. Assume that the claim holds for $i - 1$. The induction hypothesis $\pi_{i-1} \leq \pi_i$ implies that the greedy solution $Q_{i+1}$, which is chain-feasible for profile $\pi_i$, is also chain-feasible for the profile $\pi_{i-1}$. Because $Q_i$ is the greedy solution for the profile $\pi_{i-1}$, we obtain $Q_i \leq Q_{i+1}$ by Lemma 1. Therefore, $\pi_i \leq \pi_{i+1}$.                □

LEMMA 3.   *If a feasible solution $Q^*$ with $k$ paths exists, then each profile in the sequence of profiles generated by the decision procedure is bounded by the profile of $Q^*$, i.e., we have $\pi_i \leq \pi_{Q^*}$ for all $i$.*

PROOF (by induction).    Because $\pi_0$ is the empty profile, the case $i = 0$ holds trivially. Now suppose $\pi_i \leq \pi_{Q^*}$ holds for some $i$. Since the set $Q^*$ is chain-feasible for the profile $\pi_{Q^*}$ generated by itself, it is also chain-feasible for the profile $\pi_i$. Then the greedy solution $Q_{i+1}$ satisfies $Q_{i+1} \leq Q^*$ by Lemma 1, which immediately implies $\pi_{i+1} \leq \pi_{Q^*}$.     □

LEMMA 4.    *The decision procedure gives correct results and terminates after at most $n \cdot c(e_0)$ rounds.*

PROOF.    Assume first that there exists a feasible solution $Q^*$ with $k$ paths. By Lemma 3, the profile of the chain-feasible solutions computed by the algorithm always stays below the profile of $Q^*$. By Lemma 2, in each round the profile either stays the same or grows. If the profile stays the same, a feasible solution has been found by the algorithm. If the profile grows, the algorithm will execute the next round, and after finitely many rounds, a feasible solution will be found.

Now assume that the answer is "no". Again, the profile grows in each round, so there can be only finitely many rounds until the algorithm does not find $k$ paths anymore and returns "no".

We have $\sum_{j=0}^{n-1} \pi_{Q_i}(e_j) \leq n \cdot \pi_{Q_i}(e_0) \leq n \cdot c(e_0)$ for every generated profile $\pi_{Q_i}$, since profiles are non-increasing sequences and each $Q_i$ is chain-feasible. As the profile grows in each round, the number of rounds is bounded by $n \cdot c(e_0)$.     □

THEOREM 2.1.    *There is an algorithm that solves CALLCONTROL in ring networks optimally in time $O(mnc_{\min} \log m)$, where $n$ is the number of nodes in the ring, $m$ is the number of paths, and $c_{\min}$ is the minimum edge capacity (which can be assumed to be at most $m$ without loss of generality).*

PROOF.    By Lemma 4, we have a decision procedure with $n \cdot c(e_0)$ rounds to decide whether there exists a feasible solution with $k$ paths. This can be improved to $n \cdot c_{\min}$ rounds by labeling the nodes in such a way that $c(e_0)$ equals the minimum edge capacity $c_{\min}$. Each round is a pass through the $m$ given paths in greedy order, which we show in Section 3 to be implementable in linear time $O(n + m) = O(m)$ (recall that $n \leq 2m$ without loss of generality). Given the decision procedure, we can use binary search on $k$ to determine the maximum value for which a feasible solution exists with $O(\log m)$ calls of the decision procedure. Thus the total running-time of our algorithm is bounded by $O(mnc_{\min} \log m)$.     □

**3. A Linear-Time Optimal Algorithm for Call Control in Chains.**    In this section we consider the implementation of the greedy algorithm for CALLCONTROL in chain networks with arbitrary capacities that is executed in each round of the decision procedure that we used to obtain an optimal algorithm for CALLCONTROL in rings in Section 2.

The input of the greedy algorithm consists of a chain $(V, E)$ with $N$ nodes (where $N = 2n + 1$ if the chain is obtained by duplicating a ring with $n$ nodes as described in Section 2) and arbitrary edge capacities, a set of $m$ paths in the chain (each specified by its left and right endpoint), and a positive integer parameter $k$. The nodes are numbered from

0 to $N - 1$. The algorithm processes the paths in greedy order (i.e., in non-decreasing order of right endpoints) and accepts each path if it does not violate any edge capacity. It stops when either $k$ paths are accepted or all paths have been processed.

While an $O(mN)$ implementation of the greedy algorithm is straightforward, we show in the following that it can even be implemented in linear time $O(m)$, provided that a sorted list of all path endpoints is given (which can be computed in time $O(m + N)$ using bucket-sort). From the sorted list of path endpoints it is easy to determine the greedy order of the paths in linear time $O(m)$.

Let $C = \max_{e \in E} c(e)$ denote the maximum edge capacity. Without loss of generality, we can assume $C \leq m$. In the following we let the greedy algorithm run until all paths have been processed even if it accepts more than $k$ paths. In this way the greedy algorithm actually computes a maximum cardinality subset of the paths that does not violate any edge capacity (which follows from Lemma 1). Stopping the greedy algorithm as soon as $k$ paths are accepted (as needed for the ring algorithm) is then a trivial modification.

3.1. *The Algorithm by Carlisle and Lloyd for Identical Capacities*.    For the case that all edges have the same capacity $C$, a linear-time implementation of the greedy algorithm was given in [6]. We give an outline of their algorithm before we adapt it to the case of arbitrary capacities in the next section. The main idea of their algorithm is actually to compute a $C$-coloring of the accepted paths and to maintain the *leader* of each color in a data structure. A path is called the *leader* of some color if it is the greatest path in greedy order colored with that color so far. When a path $p$ is processed, the data structure is used to determine the greatest leader in greedy order that does not intersect the current path $p$. This leader is called the *best fit* leader for the path $p$, and is denoted by *leader*$(p)$. If no such leader exists, $p$ is rejected. Otherwise, $p$ is assigned the color of *leader*$(p)$ and becomes the new leader of that color.

The difficulty is in finding the best fit leader in amortized constant time, since examining all colors would take $O(\log C)$ time per path. Carlisle and Lloyd show in [6] how the union-find data structure of Gabow and Tarjan [7] can be used for this purpose. They start by computing a preferred leader *adj*$(p)$ for every path $p$, which is the greatest path in greedy order ending to the left of $p$ (see Figure 4). This can be accomplished in linear time by scanning through the paths in greedy order (which they assume to be given). Initially, every path constitutes a singleton set in the union-find data structure. Throughout the algorithm, the paths that have already been processed are contained in
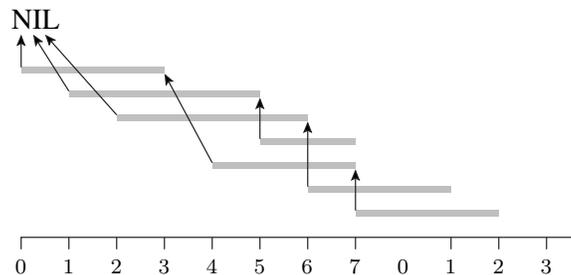


**Fig. 4.** The preferred leader *adj*$(p)$ for every path $p$.

sets consisting of a collection of consecutive paths in greedy order, and the paths that are not yet processed are in singleton sets. The representative of a set is always the smallest path in greedy order contained in that set. As the algorithm proceeds, this representative of a set will be either a leader (if the paths in the set have already been processed) or a path that has not been processed yet at that time (otherwise). Thus, when the algorithm processes a path $p$ and finds that $adj(p)$ is not a leader, the representative of the set containing $adj(p)$, which can be determined by calling find($adj(p)$), must be the best fit leader for $p$. (To handle the case that no best fit leader for $p$ exists, a fictitious leader to the left of all intervals can be introduced.)

Using this data structure, the algorithm works as follows. When a path $p$ is processed and $adj(p)$ is really a leader, then $adj(p)$ is clearly the best fit leader for $p$. Otherwise, $adj(p)$ has either been rejected or is no longer a leader. Then the operation find($adj(p)$) is used to determine the largest leader in greedy order ending no later than $adj(p)$. This is the best fit leader for $p$ in this case. If such a leader $q = leader(p)$ is found, the path $p$ is colored with the color of $q$ and the union-find data structure is updated. The path $q$ is no longer the leader of its color. Accordingly, its set is merged with the set containing the predecessor of $q$ in greedy order using a union-operation on these sets. After that, $q$ is no longer the representative for the resulting set. The fact that the path $p$ is the new leader of its color is reflected in the data structure, since $p$ continues to be the representative of its singleton set. On the other hand, if no leader is found, the path $p$ is rejected and the sets containing $p$ and the predecessor of $p$ in greedy order are merged by a union-operation. We refer to [6] for a detailed explanation why this yields a correct implementation of the greedy algorithm in the case where all edge capacities are equal.

3.2. *Adaptation to Arbitrary Capacities.* Now we adapt this approach to the case of arbitrary capacities. As a first step, we add dummy paths to the instance to fill up the $C - c(e_i)$ units of extra capacity on every edge $e_i$ as shown in Figure 5 for an example. In this new instance, we set all edge capacities equal to $C$ and compute an optimal solution among all solutions that contain all dummy paths. Removing the dummy paths from the solution yields an optimal solution for the original problem, since no edge capacity will be violated.

Before we explain how to modify the algorithm of [6] to ensure that all dummy paths are colored, we treat the dummy paths in more detail. They can be computed by scanning the chain from left to right and deciding at each node how many dummy paths should start or end here. If the edges to the left and to the right of the current node are $e_i$ and $e_{i+1}$,
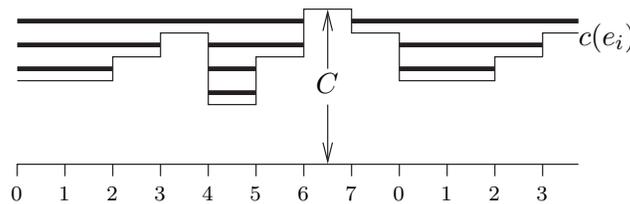

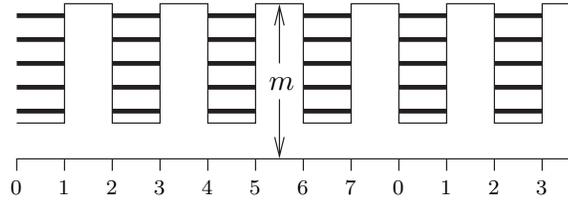
**Fig. 5.** The dummy paths for a given capacity function.

**Fig. 6.** A capacity function alternating between 1 and $m$ requires $\Omega(mN)$ dummy paths.

then $c(e_{i+1}) - c(e_i)$ dummy paths end at this node if $c(e_{i+1}) > c(e_i)$, and $c(e_i) - c(e_{i+1})$ dummy paths begin at the node otherwise.

In order to achieve an overall linear running-time, the number of dummy paths must be $O(m)$. However, there are capacity functions where $\Omega(mN)$ dummy paths are needed as shown in Figure 6.

Therefore, we introduce the following preprocessing step in order to flatten somewhat the capacity function. We scan the chain of nodes from left to right. Let $n(i)$ denote the number of original paths that have node $i$ as their left endpoint. For each edge $e_i$ we set the new capacity $c'(e_i)$ for the edge $e_i$ to the minimum of the original capacity $c(e_i)$ and $c'(e_{i-1}) + n(i)$. Hence, a decrease in the original capacity function is replicated by the new capacity function, while an increase is limited to the number of paths starting at the current node. A larger increase could not be filled out by the $n(i)$ paths anyway. We have $c'(e_i) \leq c(e_i)$ for all edges $e_i$ and that any subset of paths that is feasible for capacity function $c$ is also feasible for capacity function $c'$. To see the latter, note that the number of paths using edge $e_i$ in any feasible solution for capacity function $c$ is at most $c(e_{i-1}) + n(i)$. The new capacity function $c'$ clearly can be computed in linear time.

LEMMA 5. *With the new capacity function $c'$, the number of dummy paths added by the algorithm is $O(m)$.*

PROOF. Let us define the *total increase* of the capacity function $c'$ to be the sum of the values $\max\{c'(e_i) - c'(e_{i-1}), 0\}$ for $i = 0, \ldots, N-1$, where we take $c'(e_{-1}) = 0$. By definition of $c'$, the total increase of $c'$ is at most $m$, since every increase by 1 can be charged to a different path. Now consider the dummy paths added by the algorithm. Every dummy path ends either at an increase by 1 of the capacity function $c'$ or because the right end of the chain is reached. Therefore, there can be at most $m + C = O(m)$ dummy paths.                                                                                                      □

After preprocessing the capacity function and adding the dummy paths, we compute a maximum $C$-colorable subset of paths in which all dummy paths are colored. It is clear that then the set of colored original paths forms an optimal solution in the original chain (with capacities $c(e_i)$ or $c'(e_i)$).

We must modify the algorithm of [6] to make sure that all dummy paths are accepted and colored. First, we compute a sorted list of the endpoints of all paths, including the dummy paths, such that for every node $i$, the right endpoints of paths ending at node $i$

come before the left endpoints of paths starting at node $i$. The endpoints are processed in this order. Now the idea is to process original paths at their right endpoints and dummy paths at their left endpoints to make sure that all dummy paths are accepted and colored.

We describe the modified algorithm. First, we initialize the data structure as in [6] with $C$ *virtual* paths that serve as initial leaders for colors 1 to $C$, and a *fictitious* leader $f$. The $C$ virtual paths are considered to be located to the left of node 0, e.g., paths from node $-2$ to node $-1$, and the fictitious leader is to the left of the $C$ virtual paths, e.g., a path from $-4$ to $-3$. We extend the greedy order to these $C+1$ additional paths by taking first the fictitious leader, then the $C$ virtual paths, and then the original and dummy paths. Paths whose best fit leader is the fictitious leader will be rejected. After the initialization, all paths (including the virtual paths and the fictitious leader) are in singleton sets of the union-find data structure, and the $C$ virtual paths and the fictitious leader are considered to have been processed.

The algorithm maintains at any point the last path whose right endpoint has already been processed. This path is called *last* and is stored in a variable with the same name. As part of the initialization, we store the last virtual path in *last*.

Now the endpoints of the real paths (original paths and dummy paths) are processed in sorted order as follows. Let $x$ be the path endpoint currently being processed and let $p$ be the respective path. We distinguish two cases:

*Case* 1: *x is the left endpoint of p.*    Then we set $adj(p)$ to be the path stored in *last*, since this path would be the preferred leader for the path $p$. If $p$ is a dummy path, we want to color $p$ immediately and perform a find operation on $adj(p)$ to find $q = leader(p)$. We color $p$ with the color of $q$ and perform a union operation to merge the set containing $q$ with the set containing the predecessor of $q$ in greedy order, because $q$ is no longer a leader. If $p$ is not a dummy path, nothing needs to be done for $p$ now, because $p$ will be colored (or rejected) later when its right endpoint is processed.

*Case* 2: *x is the right endpoint of the path p.*    Then $p$ is stored in *last*, since it is now the last path whose right endpoint has already been processed. If $p$ is an original path, we want to color it now, if possible. Therefore, we perform a find operation on $adj(p)$ in order to find its leader. If such a leader $q$ is found, the color of $p$ is set to the color of $q$, and the set containing $q$ is merged with the set containing the predecessor of the path $q$ in greedy order. Otherwise, no leader was found for $p$ (i.e., the find operation on $adj(p)$ has returned the fictitious leader). The path $p$ is rejected and the set containing $p$ is merged with the set containing the predecessor of the path $p$. If $p$ is a dummy path, $p$ has already been colored at its left endpoint, so nothing needs to be done for $p$ anymore.

The union-find data structure of [7] is applicable, since the structure of the potential union operations is a tree (actually, even a chain); to see this, observe that we always merge a set containing a path $p$ with the set containing the immediate predecessor of $p$ in greedy order. Therefore, the algorithm runs in time linear in the number of all paths including the dummy paths.

We refer to the resulting algorithm as the *union-find based algorithm* and show that it is a correct implementation of the greedy algorithm (in the sense that it accepts the same subset of the original paths).

LEMMA 6. *The union-find based algorithm is a correct implementation of the greedy algorithm.*

PROOF. First, observe that the sets in the union-find data structure are maintained so that the following property is ensured: the representative of the set containing a path that has already been processed is indeed the leader with rightmost right endpoint among all leaders that do not come after the path in greedy order (or the fictitious leader if no such leader exists). Now it is easy to see that the union-find based algorithm, denoted $U$ from here on, computes a $C$-colorable subset of the paths that includes all dummy paths. We only need to show that the algorithm accepts the same original paths as the greedy algorithm. Consider the first path $p$ in greedy order for which the result of $U$ and of the greedy algorithm are different. It must be the case that $p$ is accepted by the greedy algorithm, but rejected by $U$. Algorithm $U$ processes $p$ at its right endpoint. Assume that $U$ does not accept $p$. Then the leaders of all $C$ colors must intersect $p$. Let $A$ be the set of all paths (dummy paths and original paths) that were accepted by $U$ before processing $p$. Following arguments used in [6], we will show that $A \cup \{p\}$ creates load $C + 1$ on some edge, a contradiction to the fact that the greedy solution (which contains all original paths in $A \cup \{p\}$) is feasible. Let $e$ be the rightmost edge of the leader $\ell$ whose right endpoint is as far left as possible among the $C$ leaders intersecting $p$. Assume that there is a color $i$, $1 \le i \le C$, such that no path of color $i$ contains $e$. Then the set of paths colored $i$ must contain a path $p_1$ to the left of $e$ (possibly the virtual path) and a path $p_2$ to the right of $e$. Choose such $p_1$ and $p_2$ with rightmost right endpoint and leftmost left endpoint, respectively. At the time when $p_2$ was colored by $U$, $p_1$ was the leader of color $i$ and $\ell$ was the leader of a different color. Since $\ell$ does not intersect $p_2$ and contains the edge $e$, which is strictly to the right of $p_1$, $p_1$ could not have been the best fit leader for $p_2$ (since $\ell$ is a better leader). This contradicts the fact that $U$ colored $p_2$ with color $i$. Therefore, $e$ is contained in a path of every color and also in the path $p$, implying that $A \cup \{p\}$ creates load $C + 1$ on edge $e$. Thus, we arrive at the desired contradiction. □

Summing up, our algorithm does a linear-time preprocessing of the capacity function, then adds $O(m)$ dummy paths in linear time, and finally uses an adapted version of the algorithm in [6] to run the greedy algorithm for CALLCONTROL in chains in linear time.

THEOREM 3.1. *The greedy algorithm computes optimal solutions for CALLCONTROL in chains with arbitrary edge capacities and can be implemented to run in time $O(N + m)$, where $N$ is the number of nodes in the chain and $m$ is the number of given paths.*

**4. Weighted Call Control.** In this section we consider the weighted version of the call control problem, denoted WEIGHTEDCALLCONTROL. First we show how the algorithm for the maximum weight $k$-colorable subgraph problem on interval graphs due to Carlisle and Lloyd [6] can be used to obtain optimal solutions for WEIGHTEDCALLCONTROL in chains. Then we present a simple 2-approximation algorithm for rings and give a PTAS that, due to its large running-time, is mainly of theoretical interest. Finally, we link the

complexity of the problem in rings to the bipartite exact matching problem, indicating that it may be difficult to find a deterministic polynomial-time optimal algorithm.

4.1. *Weighted Call Control in Chains.* Carlisle and Lloyd [6] presented an efficient polynomial-time algorithm for the problem of computing a maximum-weight $k$-colorable subset of a given set of weighted intervals on the real line. The algorithm is based on solving a min-cost network flow problem and takes time $O(kS(m))$, where $m$ is the number of intervals and $S(m)$ is the running-time of a shortest-path algorithm on directed graphs with positive edge weights and $O(m)$ vertices and edges. It assumes that a sorted list of interval endpoints is given as part of the input. The problem solved by Carlisle and Lloyd corresponds to WEIGHTEDCALLCONTROL in chains where all edges have capacity $k$.

In order to be able to apply their algorithm also to chains with arbitrary capacities, we simply add dummy paths of large weight. More precisely, we let $C = \max_{e \in E} c(e)$, set the capacity of every edge to $c'(e) = C$, and add a set of dummy paths with the property that every edge is contained in $C - c(e)$ dummy paths. This can be done in linear time as shown in Section 3, after preprocessing the edge capacities appropriately. Now we set the weight of each dummy path to $W + 1$, where $W$ is the sum of the weights of the original paths. Then it is clear that every optimal solution to the resulting instance $I'$ must contain all dummy paths, and the paths in the optimal solution that are not dummy paths give an optimal solution to the original instance $I$. Thus, we can apply the algorithm by Carlisle and Lloyd to the instance $I'$ in order to obtain an optimal solution to $I$ as well. Since only a linear number of dummy paths have to be added (as shown in Section 3), WEIGHTEDCALLCONTROL in chains with $N$ nodes can be solved optimally in time $O(N + C \cdot S(m))$, which includes $O(N + m)$ time for sorting the path endpoints using bucket-sort. As $C$ can be assumed to be at most $m$, the running-time is bounded by $O(N + mS(m))$.

4.2. *A 2-Approximation Algorithm for Rings.* Using the optimal algorithm for WEIGHTEDCALLCONTROL in chains as a subroutine, it is not difficult to obtain a 2-approximation algorithm for WEIGHTEDCALLCONTROL in rings. We identify an edge $e$ on the ring that has the smallest capacity $c(e) = c_{\min}$. Now take the chain obtained by removing the edge $e$ from the ring, and consider the set of paths that do not contain $e$. We can compute an optimal set of paths on the chain for this instance of WEIGHTEDCALLCONTROL in chains, as explained in the previous section. Denote this set of paths by $OPT^*_{\bar{e}}$. Next, arrange the set of paths that pass through $e$ in order of decreasing weights. Pick the first $c(e)$ paths in this order and call the resulting set $OPT^*_e$. It is easy to observe that both $OPT^*_{\bar{e}}$ and $OPT^*_e$ are feasible solutions to the original instance of WEIGHTEDCALLCONTROL in rings. Let $OPT$ be an optimal solution to that instance. Trivially, the total weight of paths in $OPT$ that are routed through $e$, denoted by $OPT_e$, satisfies $w(OPT_e) \leq w(OPT^*_e)$. Further, the weight of the paths in $OPT$ that are not routed through $e$, denoted by $OPT_{\bar{e}}$, satisfies $w(OPT_{\bar{e}}) \leq w(OPT^*_{\bar{e}})$. Hence, we have

$$
\begin{aligned}
w(OPT) &= w(OPT_e) + w(OPT_{\bar{e}}) \\
&\leq w(OPT^*_e) + w(OPT^*_{\bar{e}}) \\
&\leq 2 \cdot \max\{w(OPT^*_e), w(OPT^*_{\bar{e}})\}.
\end{aligned}
$$

By choosing the set with larger weight among $OPT_e^*$ and $OPT_{\bar{e}}^*$, we obtain a 2-approximation algorithm.

4.3. *A PTAS for Rings.*    In this section we describe a PTAS for WEIGHTEDCALLCON-TROL in rings. Let $\varepsilon > 0$ be a fixed positive constant and let $K = \lceil 2/\varepsilon \rceil$. An instance of WEIGHTEDCALLCONTROL in rings is given by a ring $G = (V, E)$ with edge capacities $c \colon E \to \mathbf{N}$ and a set $P = \{p_1, \ldots, p_m\}$ of paths with weights $w \colon P \to \mathbf{N}$. Let $c_{\min}$ denote the minimum edge capacity of the given instance, and label the edges of the ring in such a way that $e_0$ is an edge with $c(e_0) = c_{\min}$. If $c_{\min} \leq K$, we can enumerate in polynomial time all subsets $S_0 \subseteq P$ that consist of at most $c_{\min}$ paths containing the edge $e_0$; there are at most $O(m^K)$ such subsets. For each such subset $S_0$, we can use the optimal algorithm for WEIGHTEDCALLCONTROL in chains to compute a maximum weight set $S_0'$ of paths not containing $e_0$ such that $S_0 \cup S_0'$ is feasible. In the end we output the solution $S_0 \cup S_0'$ of maximum weight. As one of the enumerated sets $S_0$ must be equal to the set of paths containing $e_0$ that are accepted by the optimal solution, we obtain an optimal solution in this way.

Now consider the case that $c_{\min} > K$. First, solve the following linear programming (LP) relaxation of the problem:

$$(R) \quad \max \sum_{j=1}^{m} w_j y_j$$
$$\text{s.t.} \sum_{j=1}^{m} a_{ij} y_j \leq c_i, \qquad i = 0, \ldots, n-1,$$
$$0 \leq y_j \leq 1, \qquad j = 1, \ldots, m.$$

Here, $y_j$ represents the fraction of path $p_j$ that is accepted, $w_j = w(p_j)$ is the weight of $p_j$, $c_i = c(e_i)$ is the capacity of edge $e_i$, and $a_{ij}$ is 1 if $p_j$ contains $e_i$ and 0 otherwise.

Let $y^*$ denote the optimal solution to (R). Note that the weight of $y^*$, which equals $\sum_{j=1}^{m} w_j y_j^*$, is at least as large as $w(OPT)$, where $OPT$ is an optimal solution to the integral problem. Now we interpret $y^*$ as a fractional solution of the LP relaxation of WEIGHTEDCALLCONTROL in a chain as follows. The chain consists of $2n - 1$ edges and is obtained by duplicating every edge of the ring except $e_0$ and then "unrolling" the ring. More precisely, the middle edge of the chain is $e_0$, the left part of the chain consists of a first copy (the *left copy*) of the edges $e_1, \ldots, e_{n-1}$ of the ring, and the right part of the chain consists of a second copy (the *right copy*) of the edges $e_1, \ldots, e_{n-1}$; see Figure 7.
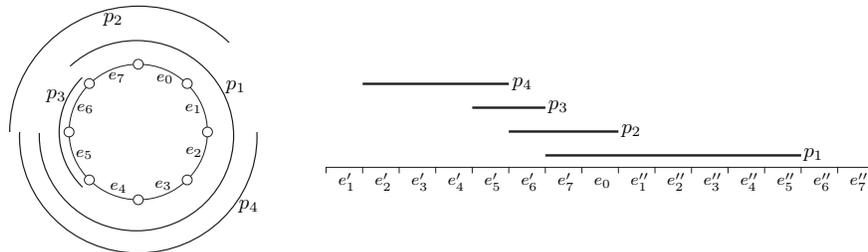


**Fig. 7.** Paths in the ring (left) are translated into paths in a chain (right) for the PTAS.

A path $p_i$ in the ring that consists of edges $e_\ell, e_{\ell+1}, \ldots, e_{n-1}, e_0, e_1, \ldots, e_r$ is translated into a path in the chain from the left copy $e'_\ell$ of $e_\ell$ to the right copy $e''_r$ of $e_r$. A path $p_i$ in the ring that does not contain $e_0$ is translated into the corresponding path in the left part of the chain. Let $p'_i$ denote the path in the chain that is obtained from $p_i$ in this way. Now, assign each edge $e$ of the chain capacity $c'(e) = \lceil \sum_{i:e \in p'_i} y^*_i \rceil$. Observe that $y^*$ is a feasible solution for the natural LP relaxation of the constructed instance of WEIGHTEDCALLCONTROL in chains. By the feasibility of $y^*$ for (R), we have that $c'(e_0) \le c(e_0)$ and $c'(e') + c'(e'') \le c(e) + 1$ for all $e \in E \backslash \{e_0\}$, where $e'$ and $e''$ are the left and right copy of $e$, respectively.

The constraint matrix of the LP relaxation of WEIGHTEDCALLCONTROL in chains is totally unimodular and, therefore, there is an optimal solution of the LP relaxation that is integral and can be computed efficiently [17]. Hence, we obtain an integral solution $\hat{y}$ to the constructed instance of WEIGHTEDCALLCONTROL in chains whose weight is at least as large as the weight of $y^*$. Now, we interpret $\hat{y}$ as a solution in the ring. Let $P^*$ be the set of paths $p_i$ such that $\hat{y}_i = 1$. Note that $P^*$ violates the edge capacities of the ring by at most 1. We proceed to show that we can remove paths from $P^*$ so that we obtain a solution that is feasible in the ring without losing too much weight.

Let $E_v$ be the set of edges whose capacity is violated by $P^*$. We compute disjoint subsets $T_1, T_2, \ldots, T_k$ of $P^*$ by repeatedly picking a subset $T_j$ of the remaining paths in $P^*$ that has maximum load 2 and the property that the union of the paths in $T_j$ contains all edges in $E_v$. (Such a subset $T_j$ can be computed easily by first adding paths containing all edges in $E_v$ and then iteratively removing paths whose edges are contained in the union of the edge sets of the other paths in $T_j$.) As $P^*$ contains at least $c_{\min} > K$ paths through each edge in $E_v$, we obtain at least $K/2$ such subsets $T_j$, thus $k \ge K/2 \ge 1/\varepsilon$. Finally, we determine $j' \in \{1, 2, \ldots, k\}$ such that the subset $T_{j'}$ has smallest weight and output the solution $Q = P^* \backslash T_{j'}$. This is a feasible solution to the original instance of WEIGHTEDCALLCONTROL in rings. As the sets $T_j$ are disjoint subsets of $P^*$ and $k \ge K/2$, the weight of $T_{j'}$ is at most a $1/k \le \varepsilon$ fraction of the weight of $P^*$, and therefore $w(Q) \ge w(P^*) - w(T_{j'}) \ge (1 - \varepsilon)w(OPT)$. Hence, we have obtained a PTAS.

THEOREM 4.1. *There is a PTAS for* WEIGHTEDCALLCONTROL *in rings*.

4.4. *Relation to Bipartite Exact Matching.* The computational complexity of WEIGHTEDCALLCONTROL in rings has not been resolved to be in *P* or *NP*-hard. However, a recent result of Hochbaum and Levin [11] shows that this problem is at least as hard as the *exact matching* problem in bipartite graphs. The exact matching problem is defined as follows. Given a graph $G = (V, E)$ and a subset $E' \subseteq E$ and a positive integer $k$, find a perfect matching of $G$ that contains exactly $k$ edges from $E'$. As pointed out by Hochbaum and Levin [11], the complexity of this problem has not been resolved to be in *P* or *NP*-hard for more than 15 years now, even if $G$ is bipartite. A randomized polynomial-time algorithm for the exact matching problem (also in the non-bipartite case) was presented by Mulmuley et al. [15].

In [11] it is shown that the bipartite exact matching problem can be reduced to MCB (bounded multicover problem, see Section 1.2). The constructed instances of MCB have

variables whose upper bounds are $u_j = 1$. We have shown in Section 1.2 that instances of MCB with polynomial upper bounds on the variables can be transformed into instances of WEIGHTEDCALLCONTROL in rings. Thus, a polynomial-time optimal algorithm for WEIGHTEDCALLCONTROL in rings would immediately give a polynomial-time optimal algorithm for MCB with polynomial upper bounds on the variables and thus also for the bipartite exact matching problem.

**5. Conclusion and Open Problems.**    The main result of this paper is a polynomial-time optimal algorithm for CALLCONTROL in ring networks. CALLCONTROL in rings is significantly more general than the maximum edge-disjoint paths problem for rings and appears to be close to the maximum $k$-colorable subgraph problem for circular-arc graphs, which is *NP*-hard. Therefore, we find it interesting to see that CALLCONTROL in rings is still on the "polynomial side" of the complexity barrier. Besides its applications in call admission control for communication networks, the algorithm can also be used to compute optimal solutions for various cyclical scheduling problems. We have shown that the algorithm can be implemented efficiently, and as a by-product we have obtained a linear-time implementation of the greedy algorithm that solves CALLCONTROL in chains optimally. For WEIGHTEDCALLCONTROL in rings, we have presented a 2-approximation algorithm and a PTAS and pointed out that solving the problem optimally is at least as difficult as the bipartite exact matching problem. The latter problem is known to have a randomized polynomial-time algorithm (even in the non-bipartite case) [15], but to date its complexity has not been established to be in *P* or *NP*-hard according to [11].

Our results lead to several open questions for future research. We do not know whether WEIGHTEDCALLCONTROL in rings is *NP*-hard or there is a (possibly randomized) polynomial-time optimal algorithm for it.

For the CALLCONTROLANDROUTING problem, where only the path endpoints are given and the algorithm must determine the routing of the accepted requests, an approximation algorithm with an additive approximation guarantee for rings is presented in [2]. The complexity of that problem (polynomial or *NP*-hard?) is also unresolved, and we are not aware of any results about the weighted version (for which a 2-approximation algorithm can be obtained using standard techniques).

<div align="center">

**References**

</div>

[1]  R. Adler and Y. Azar. Beating the logarithmic lower bound: randomized preemptive disjoint paths and call control algorithms. In *Proceedings of the* 10*th Annual ACM–SIAM Symposium on Discrete Algorithms* (*SODA* '99), pages 1–10, 1999.

[2]  R. S. Anand and T. Erlebach. Routing and call control algorithms for ring networks. In *Proceedings of the* 8*th International Workshop on Algorithms and Data Structures* (*WADS* 2003), LNCS 2748, pages 186–197, 2003.

[3]  M. J. Atallah, D. Z. Chen, and D. T. Lee. An optimal algorithm for shortest paths on weighted interval and circular-arc graphs, with applications. *Algorithmica*, 14:15–26, 1995.

[4]  A. Blum, A. Kalai, and J. Kleinberg. Admission control to minimize rejections. In *Proceedings of the* 7*th International Workshop on Algorithms and Data Structures* (*WADS* 2001), LNCS 2125, pages 155–164, 2001.

[5]   A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.

[6]   M. C. Carlisle and E. L. Lloyd. On the *k*-coloring of intervals. *Discrete Applied Mathematics*, 59:225–235, 1995.

[7]   H. Gabow and R. Tarjan. A linear-time algorithm for a special case of disjoint set union. *Journal of Computer and System Sciences*, 30(2):209–221, 1985.

[8]   J. A. Garay, I. S. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient on-line call control algorithms. *Journal of Algorithms*, 23:180–194, 1997.

[9]   M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM Journal on Algebraic and Discrete Methods*, 1(2):216–227, 1980.

[10]  M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.

[11]  D. S. Hochbaum and A. Levin. Cyclical scheduling and multi-shift scheduling: complexity and approximation algorithms. Manuscript, UC Berkeley, June 2003.

[12]  D. S. Hochbaum and P. A. Tucker. Minimax problems with bitonic matrices. *Networks*, 40:113–124, 2002.

[13]  I. A. Karapetian. On the coloring of circular arc graphs. *Journal of the Armenian Academy of Sciences*, 70(5):306–311, 1980 (in Russian).

[14]  S. Leonardi. On-line network routing. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms*: *The State of the Art*, LNCS 1442, pages 242–267. Springer-Verlag, Berlin, 1998.

[15]  K. Mulmuley, U. V. Vazirani, and V. V. Vazirani. Matching is as easy as matrix inversion. *Combinatorica*, 7(1):105–113, 1987.

[16]  S. Plotkin. Competitive routing of virtual circuits in ATM networks. *IEEE Journal of Selected Areas in Communications*, 13(6):1128–1136, August 1995.

[17]  A. Schrijver. *Combinatorial Optimization*: *Polyhedra and Efficiency*. Springer, Heidelberg, 2003.

[18]  G. Wilfong and P. Winkler. Ring routing and wavelength translation. In *Proceedings of the Ninth Annual ACM–SIAM Symposium on Discrete Algorithms* (*SODA* '98), pages 333–341, 1998.