In: Proceedings of the 29th International Colloquium on Automata,Languages, and Programming ICALP 2002, Lecture Notes in Computer Science 2380, July 2002, pp. 788–799. ©Springer-Verlag

Call Control in Rings*

Udo Adamy[1], Christoph Ambuehl[1], R. Sai Anand[2,**], and Thomas Erlebach[2]

[1] Institute for Theoretical Computer Science, ETH Zürich, 8092 Zürich, Switzerland.
{adamy|ambuehl}@inf.ethz.ch

[2] Computer Engineering and Networks Laboratory, ETH Zürich, 8092 Zürich, Switzerland.
{anand|erlebach}@tik.ee.ethz.ch

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 call control problem in chains optimally.

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 *call admission control problem*, where the task is to determine which of the requests in a given set of connection requests to accept or reject so as to optimize some objective, e.g., the number of accepted requests. In this paper, we consider a call admission control problem in ring networks and prove that it can be solved optimally by an efficient polynomial-time algorithm. The ring topology is a fundamental network topology that is frequently encountered in practice. As an additional application of our algorithm, we show that it can also be used to compute optimal solutions to periodic scheduling problems with rejection.

Problem Definition and Applications. 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 $c : E \to \mathbb{N}$ and a multi-set P of m paths in (V, E). 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 multi-set $Q \subseteq P$ such that for every edge $e \in E$, the number of paths in Q that contain e is at most $c(e)$.

* Research partially supported by the Swiss National Science Foundation.

** Supported by the joint Berlin/Zurich graduate program Combinatorics, Geometry, and Computation (CGC), financed by ETH Zurich and the German Science Foundation (DFG).

Such a multi-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. Whenever we talk about a set of paths in the following, we allow that the set is actually a multi-set, i.e., that it can contain several instances of the same path. In this paper we deal with CALLCONTROL mainly in ring networks. A *ring network* 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 $c(e_i)$.

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 [13]. Thus, call admission control in such networks can be modeled as CALLCONTROL with all edge capacities equal to w.

Furthermore, it should be noted that problems related to call admission 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.

We briefly discuss an application in periodic scheduling. Without loss of generality, we 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 10am to 5pm and another task from 3pm to 2am 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. 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.

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 [9]. Circular-arc graphs can be recognized efficiently [5]. For a given circular-arc graph, a maximum clique or a maximum independent set can be computed in polynomial time [9]. Coloring a circular-arc graph with the minimum number of colors is NP-hard [8]. A coloring with at most $1.5\,\omega$ colors always exists and can be computed efficiently [10], where ω is the size of a maximum clique in the graph. Concerning our CALLCONTROL problem, we note that the special case where all edges have capacity 1 is equivalent to the maximum 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 CALLCONTROLANDROUTING. The special case of CALLCONTROLANDROUTING where all edges have capacity 1 is called the *maximum edge-disjoint paths* problem (MEDP). We refer to [3, Chapter 13] and [11, 12] for surveys on on-line algorithms for call control problems and mention only some of the known results here.

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 [4]. 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 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 [7], 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 [2], 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.

2 Preliminaries

Let $P = \{p_1, \ldots, p_m\}$ denote the given set of m paths, each connecting two nodes in the ring network. Every $p_i \in P$ is an ordered pair of nodes $p_i = (s_i, t_i)$ from V^2 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 clockwise direction. 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 subset $Q \subseteq P$ is called *feasible* if the ringload does not exceed the capacity of any edge, 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 going 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, the *tail* from node 0 to its target node. To simplify the explanation we introduce a node n and identify it with node 0. From now on, the paths with target node 0 are treated as if they end at node n. Thus we have the characterization $P_1 = \{p_i \in P : s_i < t_i\}$ and $P_2 = \{p_i \in P : s_i > t_i\}$. Note that $P = P_1 \cup P_2$.

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. Paths with equal target nodes are ordered arbitrarily. We call this ordering *greedy*. In the example of Fig. 1(a) the paths $p_1, \ldots, p_6$ are in greedy order. The solid paths $p_1, \ldots, p_4$ are in P_1. P_2 consists of the dotted paths p_5 and p_6.

The algorithm considers a chain of $2n$ edges consisting of two copies of the ring glued together. The chain begins with the first copy of e_0 and ends with the second copy of e_{n-1}, see Fig. 1(b). The tails of the P_2-paths are in the second copy of the ring, while the P_1-paths and the heads of the P_2-paths are in the first copy. Note that the greedy order of the paths corresponds to an ordering by right endpoints in this chain.

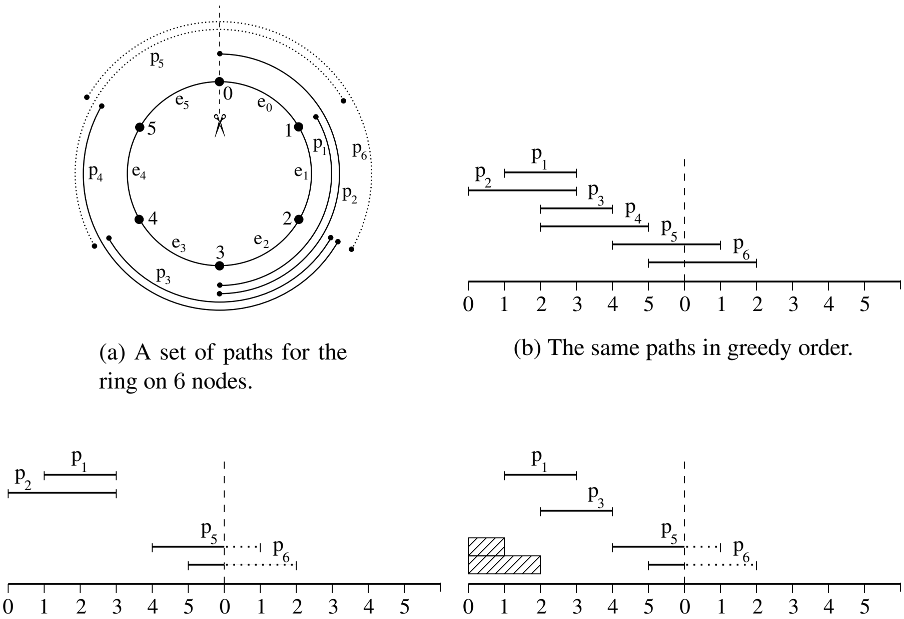
For a given set Q of paths, we define $L_1(Q, e_i)$ and $L_2(Q, e_i)$ to be the load of the paths in Q on the first copy of e_i and their load on the second copy of e_i, respectively. Thus, the paths in P_1 and the heads of the paths in P_2 contribute to the load values $L_1(P, e_i)$ of the first copy of the ring. The tails of the P_2-paths determine the load values $L_2(P, e_i)$. The ringload L is simply the sum of L_1 and L_2. With this definition of L_2, we can introduce the central notion of profiles.

Definition 1 (Profiles). *Let Q be a set of paths. The profile π 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,*

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

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 π and π' we have $\pi \leq \pi'$, iff $\pi(e_i) \leq \pi'(e_i)$ for all edges $e_i \in E$.

A set Q of paths is called *chain-feasible* if it does not exceed the capacity of any edge in this 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 $L_2(Q, e_i) \leq c(e_i)$ for all $e_i \in E$. It is called *chain-feasible for (starting) profile π* 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 π_Q.



(a) A set of paths for the ring on 6 nodes.

(b) The same paths in greedy order.

(c) The candidate set Q_1 of the first round. Its profile is dotted.

(d) The feasible solution Q_2 found in round 2.

Fig. 1. The decision procedure. Is there a feasible solution with 4 paths?

3 The Algorithm

The goal of the algorithm is to find a maximum sized feasible subset of paths in P. 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 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 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 both 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 the 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.

We illustrate the decision procedure at the example in Fig. 1. Let the capacities be $c(e_i) = 2$ for every edge $e_0, \ldots, e_5$. We ask for a feasible solution consisting of $k = 4$ paths. The paths are always processed in the greedy order, which is shown in Fig. 1(b). In the first round the paths p_1 and p_2 are accepted. The paths p_3 and p_4 are rejected, because they violate the capacity constraint of the edge e_2 after the paths p_1 and p_2 have been accepted. The paths p_5 and p_6 are both accepted to form the candidate set $Q_1 = \{p_1, p_2, p_5, p_6\}$ of 4 paths shown in Fig. 1(c). The profile of Q_1 is 2 for the edge e_0, 1 for the edge e_1, and 0 elsewhere. Q_1 is not feasible because $L(Q_1, e_0) = 3$ exceeds the capacity $c(e_0) = 2$.

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 $Q_2 = \{p_1, p_3, p_5, p_6\}$ illustrated in Fig. 1(d). The path p_2 is rejected this time, because both edges e_0 and e_1 are saturated by the profile of Q_1 and the path p_1. The path p_4 is rejected for the same reason as before. The profile of Q_2 is again 2 for the edge e_0, 1 for the edge e_1, and 0 elsewhere. Since the resulting profile π_{Q_2} is equal to the starting profile π_{Q_1}, Q_2 is a feasible solution of size 4. The procedure stops.

4 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 π_0 is the empty profile we start with, and Q_i denotes the chain-feasible solution computed in round i. We set the profile $\pi_i := \pi_{Q_i}$.

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 ith path chosen by the greedy algorithm. If A and B are two chain-feasible solutions, we write $A \leq B$, iff $a_i \leq b_i$ 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 ith path in A with the ith 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 π' is also chain-feasible for profile π. In the following, we call a solution A that is chain-feasible for profile π *minimal* if for any other solution B that is chain-feasible for π and has the same cardinality as A, we have $A \leq B$.

Lemma 1 (Optimality of greedy algorithm). *Let π be some starting profile. If there exists a solution of size k that is chain-feasible for profile π, 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 π 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, look at 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 π_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 π_i, is also chain-feasible for the profile π_{i-1}. Because Q_i is the greedy solution for the profile π_{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) Since π_0 is the empty profile, the case $i = 0$ holds trivially. Now suppose $\pi_i \leq \pi_{Q^*}$ holds for some i. Because Q^* is chain-feasible for π_{Q^*}, it is also chain-feasible for π_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 says "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 π_{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 1. *There is a polynomial-time algorithm that computes an optimal solution for* CALLCONTROL *in rings.*

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. Each round is a pass through the m given paths in greedy order, which can obviously be implemented in polynomial time. The number of rounds is polynomial as well, since we can assume without loss of generality that $c(e_0) \leq m$.

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. □

5 Efficient Implementation

In this section, we discuss how the algorithm can be implemented efficiently and analyze the worst-case running-time. Let an instance of CALLCONTROL be given by m paths in a ring with n nodes. Each path is specified by its counterclockwise and clockwise endnode. We assume $n \leq 2m$ since every node that is not an endpoint of a path can be removed. A sorted list of all path endpoints can be computed in time $O(m + n)$ using bucketsort, and it suffices to do this once at the start of the algorithm. From this list it is easy to determine the greedy order of the paths in linear time.

First 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. 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 $O(m)$.

5.1 Implementation of the Greedy Algorithm for Chains

The input of the greedy algorithm consists of a chain with $N = 2n + 1$ nodes and arbitrary edge capacities, a set of m paths in the chain, and a parameter k. The algorithm

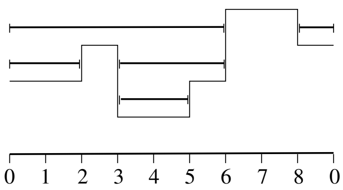


Fig. 2. The dummy paths for a given capacity function.

processes the paths in greedy order 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.

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 assume that 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 [4]. Stopping the greedy algorithm as soon as k paths are accepted is then a trivial modification.

For the case that all edges have the same capacity C, a linear-time implementation of the greedy algorithm was given in [4]. The main idea of their algorithm is to actually compute a C-coloring of the accepted paths and to maintain the *leader* for each color (the greatest path in greedy order colored with that color so far) in a data structure. When a path p is processed, the rightmost (greatest in greedy order) leader not intersecting p, denoted by $leader(p)$, is determined. 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 union-find data structure of [6] is used to compute leaders in amortized constant time. For this purpose, each path p has a preferred leader $adj(p)$, which is the greatest path in greedy order ending to the left of p. When p is processed and $adj(p)$ is really a leader, the correct leader for p is $adj(p)$. Otherwise, $adj(p)$ has either been rejected or is no longer a leader, and an operation $\text{find}(adj(p))$ is used to determine the rightmost leader ending no later than $adj(p)$, which is the correct leader for p. If such a leader is found, p is colored with the color of that leader and the sets containing $leader(p)$ and $pred(leader(p))$ are merged, where $pred(q)$ denotes the last path before q in the greedy order. If no leader is found, p is rejected and the sets containing p and $pred(p)$ are merged. We refer to [4] for a detailed explanation why this yields a correct implementation of the greedy algorithm.

In order to adapt this approach to the case of arbitrary capacities, 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 Fig. 2 for an example. After setting all edge capacities equal to C, we compute an optimal solution containing all dummy paths. Removing them from the solution yields an optimal solution for the original problem. We will show later how to modify the algorithm of [4] to ensure that all dummy paths are colored. The dummy paths are 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}, then $c(e_{i+1}) - c(e_i)$ dummy paths end at the 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 a linear running-time, the number of dummy paths should be $O(m)$. However, there are capacity functions where $\Omega(mn)$ dummy paths are needed (e.g., capacities alternating between 1 and m). Therefore, we introduce the following preprocessing step in order to somewhat flatten 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. 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' can clearly 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 because of an increase by 1 of 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 [4] to make sure that all dummy paths are accepted and colored. We assume that all paths including dummy paths are given as a sorted list of their endpoints such that for every node i, the right endpoints of paths ending at i come before the left endpoints of paths starting at 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 give a rough sketch of the resulting algorithm, omitting some details such as the initialization of the union-find data structure (which is the same as in [4]). 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.

Let x be the path endpoint currently being processed and p the respective path. First, consider the case that x is the left endpoint of p. Then we set $adj(p)$ to be the path stored in *last*. 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 $pred(q)$. If p is not a dummy path, nothing needs to be done for p now, because p will be colored later when its right endpoint is processed.

Now, consider the case that x is the right endpoint of 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 $pred(q)$; otherwise, p is rejected and the set containing p is merged with the set containing $pred(p)$. 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 [6] is applicable, since the structure of the potential union operations is a tree (actually, even a chain). Therefore, the algorithm runs in time linear in the number of all paths including the dummy paths. The arguments for proving that this gives a correct implementation of the greedy algorithm are similar to the ones given in [4] and are omitted here. Furthermore, it can be shown similar to Lemma 1 that the computed solution is optimal.

Summing up, the algorithm does a linear-time preprocessing of the capacity function, then adds $O(m)$ dummy paths in linear time, and then uses an adapted version of the algorithm in [4] to run the greedy algorithm in time linear in the number of paths.

Theorem 2. *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.*

5.2 Analysis of Total Running Time for Rings

An instance of CALLCONTROL in ring networks is given by a capacitated ring with n nodes and m paths in the ring. To implement the algorithm of Sect. 3, we use binary search on k to determine the maximum value for which a feasible solution exists. This amounts to $O(\log m)$ calls of the decision procedure. In each call of the decision procedure, the number of rounds is bounded by $n \cdot c(e_0)$ according to Lemma 4. This can be improved to $n \cdot c_{\min}$ by labeling the nodes such that $c(e_0)$ equals the minimum edge capacity $c_{\min}$. Each round consists of one execution of the greedy algorithm, which takes time $O(m)$ as shown in Sect. 5.1. Thus the total running-time of our algorithm is bounded by $O(mnc_{\min} \log m)$.

Theorem 3. *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.*

A minor improvement in the number of calls of the decision procedure may be obtained on certain instances as follows. We first run the greedy algorithm of Sect. 5.1 on the paths in P_1. This yields an optimal feasible subset Q of P_1. Let $t = |Q|$. Then we know that the size of an optimal feasible subset of P lies in the interval $[t, t + \min\{|P_2|, c(e_0), c(e_{n-1})\}]$. The number of calls of the decision procedure is reduced to $O(\log \min\{|P_2|, c(e_0), c(e_{n-1})\})$.

6 Conclusion and Open Problems

We have presented an algorithm for CALLCONTROL in ring networks that always computes an optimal solution in polynomial time. 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 solve periodic scheduling problems with rejection. Furthermore, the algorithm can be implemented efficiently, and as a by-product we obtain a linear-time implementation of the greedy algorithm that solves CALLCONTROL in chains optimally.

These results lead to some open questions for future research. First, one could consider a weighted version of CALLCONTROL where each request has a certain profit and the goal is to maximize the total profit of the accepted requests. Second, one could try to tackle the version of CALLCONTROL where the paths for the accepted requests can be determined by the algorithm. For both problem variants, we do not yet know whether they can be solved optimally in polynomial time as well. We remark that the weighted version of CALLCONTROL in chains can be solved in polynomial time by adapting the approach based on min-cost network flow of [4].

References

1. R. Adler and Y. Azar. Beating the logarithmic lower bound: Randomized preemptive disjoint paths and call control algorithms. In *Proceedings of the 10th Annual ACM–SIAM Symposium on Discrete Algorithms SODA'99*, pages 1–10, 1999.
2. A. Blum, A. Kalai, and J. Kleinberg. Admission control to minimize rejections. In *Proceedings of the 7th International Workshop on Algorithms and Data Structures (WADS 2001)*, LNCS 2125, pages 155–164, 2001.
3. A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
4. M. C. Carlisle and E. L. Lloyd. On the k-coloring of intervals. *Discrete Applied Mathematics*, 59:225–235, 1995.
5. E. M. Eschen and J. P. Spinrad. An $O(n^2)$ algorithm for circular-arc graph recognition. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms SODA'93*, pages 128–137, 1993.
6. 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.
7. 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.
8. M. R. Garey, D. S. Johnson, G. L. Miller, and C. H. Papadimitriou. The complexity of coloring circular arcs and chords. *SIAM J. Algebraic Discrete Methods*, 1(2):216–227, 1980.
9. M. C. Golumbic. *Algorithmic Graph Theory and Perfect Graphs*. Academic Press, New York, 1980.
10. I. A. Karapetian. On the coloring of circular arc graphs. *Journal of the Armenian Academy of Sciences*, 70(5):306–311, 1980. (in Russian)
11. S. Leonardi. On-line network routing. In A. Fiat and G. J. Woeginger, editors, *Online Algorithms: The State of the Art*, LNCS 1442. Springer-Verlag, Berlin, 1998.
12. S. Plotkin. Competitive routing of virtual circuits in ATM networks. *IEEE Journal of Selected Areas in Communications*, 13(6):1128–1136, August 1995.
13. 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.