Competitive Analysis of Call Admission Algorithms that Allow Delay

Anja Feldmann Bruce Maggs Jiří Sgall[1]

Daniel D. Sleator Andrew Tomkins

January 16, 1995

CMU-CS-95-102

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

[1]Institute of Computer Science, Hebrew University, Jerusalem 91904; on leave from Mathematical Institute, AV ČR, Žitná 25, 115 67 Praha 1, Czech Republic.

This research is sponsored in part by an NSF National Young Investigator Award, No. CCR-9457766, NSF grant no IRI-9314969, and by the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, and the Advanced Research Projects Agency (ARPA) under grant F33615-93-1-1330. The US Government is authorized to reproduce and distribute reprints for Government purposes, notwithstanding any copyright notation thereon. Views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of NSF or Wright Laboratory or the United States Government.

Keywords: call admission, high-speed networks, on-line algorithms, competitive analysis, Greedy algorithm

Abstract

This paper presents an analysis of several simple on-line algorithms for processing requests for connections in distributed networks. These algorithms are called call admission algorithms. Each request comes with a source, a destination, and a bandwidth requirement. The call admission algorithm decides whether to accept a request, and if so, when to schedule it and which path the connection should use through the network. The duration of the request is unknown to the algorithm when the request is made. We analyze the performance of the algorithms on simple networks such as linear arrays, trees, and networks with small separators. We use three measures to quantify their performance: makespan, maximum response time, and data-admission ratio. Our results include a proof that greedy algorithms are $\Theta(\log n)$-competitive with respect to makespan on n-node trees for arbitrary durations and bandwidth, a proof that on an n-node tree no algorithm can be better than $\Omega(\log \log n / \log \log \log n)$-competitive with respect to makespan, and a proof that no algorithm can be better than $\Omega(\log n)$-competitive with respect to call-admission and data-admission ratio on a linear array, if each request can be delayed for at most some constant times its (known) duration.

1 Introduction

1.1 Motivation

Recent advances in network technology suggest that, in the future, various classes of traffic — voice, video, and data — will be transmitted on the same high-speed digital network. A user of such a network initiates a call by submitting a request to a service provider for a connection between two nodes of the network. For example, a user might submit a request to hold a video conference between Pittsburgh and San Francisco. In this request the user specifies the required quality of service such as bandwidth, but the user might be unable to specify the duration of the connection. The user hopes to obtain the desired connection within a reasonable amount of time.

The service provider operates a network in which nodes are connected with links of varying bandwidth capacities. For the service provider to guarantee service, shared resources such as bandwidth on links and buffer capacity at nodes have to be reserved along some path between the terminal nodes of each connection. Since several calls might require the same network resources, the service provider uses a call admission algorithm to decide which requests to grant, when to schedule them, and which paths to use. Call admission is one of the congestion control mechanisms that guarantees the necessary resource to already established calls.

Unlike traditional voice circuits in which the bandwidth requirement is small, some applications, like video, can have high bandwidth requirements. Given such bandwidth requirements, one can expect to see only a small number of connections sharing the same physical link, and hence the traffic on such a link cannot be modeled accurately by statistical means. Traditional methods of analyzing congestion control schemes rely on statistical models to forecast future requests and therefore have restricted use if applied to high-speed networks. Competitive analysis [ST85], i.e., comparing the solution that an algorithm produces on any sequence of requests to the best solution that any algorithm could produce on that sequence of requests, does not rely on forecasting future events and as such is a realistic alternative method.

1.2 The model

The communication network is modeled as an undirected graph with capacities on its edges. *Requests* (we use the terms call and request interchangeably) arrive at various times. Each is specified by a pair of terminal nodes, and the bandwidth required. The job of the call admission algorithm is to decide if *and when* the call should be placed, and which path through the network should be used for the call. Note that we allow the call admission algorithm to delay the start of a call, rather than simply making a binary decision to accept or reject it. Obviously the total bandwidth of the calls on a link in the network cannot exceed its capacity. Once a call is accepted, it remains in place until completion – preemption is not allowed.

The performance of a call-admission algorithm is evaluated by the quality of the schedules that it produces. In this paper we consider the following measures of schedule quality. The *makespan* of a schedule is the total time required to complete all the calls. The *maximum response time* is the maximum time between when a request arrives and when it is scheduled to

begin. For call admission algorithms that reject some requests, we consider the *data-admission ratio*, which is the ratio of the amount of data (bandwidth times duration times distance between the terminal nodes) in the requests that are scheduled to the total amount of data in all of the requests, and the *call-admission ratio*, which is the ratio of the number of requests scheduled to the total number of requests.

All of these performance measures quantify important aspects of the behavior of the algorithms. Obviously the maximum response time is important to the user of the network. The goal of the service provider is to achieve the highest network utilization possible. If all requests are satisfied, then the schedule with the shortest makespan has the highest network utilization. On the other hand if some requests are not satisfied, call-admission and data-admission ratios capture the network utilization.

We characterize the performance of an on-line call admission algorithm in terms of its competitive ratio with respect to some measure. The *competitive* ratio of an on-line algorithm is the maximal ratio over all inputs of the performance of a schedule computed by the algorithm to the performance of an optimum schedule. An algorithm is *k-competitive* if it has a competitive ratio of at most k.

One way in which our work differs from previous analysis is that we allow our call admission algorithms to delay calls. In practice, some delay is almost always acceptable, and in fact delay is routinely encountered when making phone calls, setting up telnet connections, or sending email messages. Also, some sources of delay, such as the latency, or transit time, between the terminals of a connection are outside of the control of a call admission algorithm. Thus, since some delay is likely to be inevitable, it makes sense to allow a call admission algorithm a little leeway.

In order to investigate possible advantages of delaying calls, we study two variations of the basic model: the first places no limit on the delay allowed before a call is scheduled, while the second limits the maximum allowed delay for each request to some multiple of its own duration. While the duration of the request can be unknown for the first model it must be known for the second one.

Except for the case when the delay is bounded, in our model the duration of a request is unknown when it is initiated, and can only be determined by scheduling the request and observing how long it takes to complete. This model differs from previously considered models for the on-line call admission problem which were typically of the following form: the existence of a request is unknown until a certain arrival time, at which point all parameters of this request are completely specified.

1.3 Our results

In Section 2 we study whether the competitive ratio with respect to the data-admission ratio improves if a request can be delayed by an amount of time that is proportional to its duration. We allow a request of known duration d to be scheduled within time cd after its arrival, for some constant $c > 0$. We show that for any constant $c \geq 0$, the competitive factor of any randomized on-line algorithm is at least $\Omega(\log n)$, even for the linear array, both for the data-admission

ratio and for the call-admission ratio. This is true even if the on-line schedule is compared to an off-line schedule that is not allowed to delay requests at all. Hence, delaying the calls for a limited time does not help at all. This lower bound generalizes the $\Omega(\log n)$ lower bounds of [ABFR94, LT94], which consider the case in which no delay is allowed (i.e., $c = 0$).

In view of this lower bound, in the rest of the paper we allow a request to be delayed for an arbitrary period of time. Two closely related performance measures, makespan and maximum response time are considered. When the network is overloaded, maximizing the network utilization by minimizing the makespan has the benefit of keeping periods of overload as short as possible and therefore helps the network to return to normal operation as quickly as possible. We analyze algorithms with respect to makespan in Sections 3 through 5. In Section 7 we provide an algorithm-independent measure of when the network is overloaded and show that for certain algorithms the response time of any request is at most as long as the (length of the) period of overload in which the request was issued.

We note that algorithms that are competitive with respect to makespan when all calls appear at once, called *batch-style algorithms*, are also competitive when calls arrive over time, called *fully on-line algorithms*, with at most a factor of 2 change in the competitive ratio. This result follows immediately from a theorem of Shmoys, Wein, and Williamson [SWW91]. In Section 6, we strengthen this result by showing that by placing some mild restrictions on the type of algorithms allowed, a k-competitive batch-style algorithm can be converted into a $(k + 1)$-competitive fully on-line algorithm. An algorithm that satisfies these restrictions is called *mergeable*.

The most basic call admission algorithm simply schedules each call as soon as it can satisfy its requirements. Algorithm GREEDY (described and analyzed in Section 3) is the batch-style version of this. (GREEDY is still an on-line algorithm since the requests have unknown durations.) In addition, it is mergeable, so any bound on its competitive factor with respect to makespan immediately translates to the fully on-line variant of it. We analyze the competitive factor of GREEDY with respect to makespan on linear arrays, trees, and networks with small separators. In Section 3.1.1 we describe GREEDY on tree networks and in Section 3.2 we show that it is $O(\log n)$-competitive on any n-node tree with unit edge capacity, provided that there is an upper bound of $1 - 1/c$ on the bandwidth of any call, for some fixed $c > 0$. We give a simple modification of GREEDY that is $O(\log n)$-competitive on any n-node tree for arbitrary bandwidth requirements. This modification is unavoidable, since, as we show in Section 3.3, GREEDY may perform arbitrarily badly if arbitrary bandwidth requirements are allowed. In the same section we show that our analysis of GREEDY is tight since it cannot be better than $\Omega(\log n)$-competitive on binary trees even if all bandwidth requirements are 1. Addressing the routing problem, we show how to generalize GREEDY to networks with small separators by providing a way to chose paths for the requests that are reasonable with respect to the separators. In Section 3.2 we prove the generalized upper bound on the competitive ratio.

In Section 4 we prove an $\Omega(\log \log n / \log \log \log n)$ lower bound for any deterministic on-line call admission algorithm on a tree network with respect to makespan. This non-constant lower bound confirms that there is a fundamental difference in call admission on a linear array network and a tree network.

Considering several restricted cases provides useful insight into the combinatorial structure of the call admission problem. Using the relationship of call admission to on-line coloring and to dynamic storage allocation we show (Section 5.1) that the competitive ratio of GREEDY on a linear array network is between 4.4 and 25.8 if each edge has unit capacity, and each request requires unit bandwidth. Our analysis of GREEDY leaves a rather large gap between the upper and lower bounds for its competitive ratio for the linear array network. In Section 5.2 we describe another mergeable, batch-style algorithm, called STEADY. We show that STEADY is c-competitive if the bandwidth required by each request is between $1/c$ and $1/2$.

All of our competitive ratios are independent of the maximum duration and the ratio of the largest to the smallest bandwidth requirement.

1.4 Related work

Awerbuch et al. [ABFR94] present on-line call-admission algorithms for trees. In their model, requests include known bandwidth, duration and benefit, and must be scheduled immediately or rejected. They present a general technique called "classify and randomly select", in which the algorithm randomly selects a bandwidth b, duration d, and benefit f, all of which are powers of 2. The algorithm then restricts itself to calls that have bandwidth between b and $2b$, duration between d and $2d$, and benefit between f and $2f$, rejecting all other calls. Using tree separators, they apply a similar technique to schedule the restricted set of calls with $O(\log n)$ competitive ratio, with respect to total benefit. The total competitive ratio is therefore $O(\log n \log M \log T \log F)$, and the authors give a lower bound of $\Omega(\log n \log M \log T)$. Yet, all algorithms designed using the "classify and randomly select" paradigm seem unsatisfactory to a real user of a network, even though they are competitive.

Another possible problem with such a model is that the sequence of requests presented to a call admission algorithm is highly dependent on the way that the algorithm responds to requests. For example, whenever a call admission algorithm immediately rejects a request, it is likely to see that request immediately reissued by a user. Although we may be able to use competitive analysis to prove that such an algorithm performs close to optimally on such a sequence, and indeed on any sequence presented to it, if the algorithm itself encourages worst-case sequences (with respect to, say, the achievable call-admission ratio), then the use of competitive analysis is counterproductive. Therefore another reason to consider algorithms that may delay calls is that in practice, call admission algorithms that reject fewer (if any) requests seem less likely to encourage worst-case sequences of requests and are more likely to be acceptable to the user of the network.

Lipton and Tomkins [LT94] study the problem of scheduling a single resource for requests with known duration. As in [ABFR94], requests must be scheduled immediately or rejected. The goal is to keep the resource busy for as much time as possible. They show a $\log^{1+\epsilon}(T)$-competitive algorithm, in which T is the ratio between longest and shortest requests, and need not be known in advance. They also show an $\Omega(\log T)$ lower bound on the competitive ratio of any algorithm.

Awerbuch, Gawlick, Leighton, and Rabani [AGLR94] focus (mainly) on the problem of

designing a call admission algorithm for the case in which network links have unit capacity, and requests have unit bandwidth requirements and infinite durations. They present, among other things, an algorithm that is $\Theta(\log(d))$-competitive, with respect to the call-admission ratio, for tree networks of radius d, and an algorithm that is $O(\log n \log\log n)$-competitive for meshes.

Garay et al. [GGK+93] give algorithms for linear array networks, under various benefit models. They assume that durations of calls are known in advance, and in most of their models they assume that all bandwidths are 1. Awerbuch et al. [AAPW94] study calls of unknown durations, but do so in the context of relative load on the network, in a model that allows calls to be rerouted a limited number of times. Awerbuch et al. [AAP93] give $O(\log n)$-competitive algorithms for general topologies, but do not allow calls to require more than a $1/\log n$ fraction of the available bandwidth of a single link.

2 Limiting the maximum delay

In this section we allow a request to be delayed for some bounded time but not indefinitely. We suppose that a request of duration d has to be scheduled within time cd after its arrival for some constant $c \geq 0$; if it is not scheduled, it is rejected and we do not succeed in sending the data associated with it. The duration of a request is known as soon as it arrives.

In this case it is not interesting to consider makespan as a performance measure, since it is obviously bounded in terms of the arrival times of jobs, their durations, and their maximal allowed delay. On the other hand, not all calls are scheduled, and thus it is important to consider both the call-admission ratio (the number of scheduled requests divided by the total number of requests) and the data-admission ratio (the total amount of data in the scheduled requests divided by the total amount of data in all requests).

The case in which each request has to be satisfied immediately or rejected ($c = 0$ in our notation), was studied in [ABFR94, LT94] where it was proved that, even for a single edge, the competitive factor of any randomized algorithm is at least $\log n$, i.e., any algorithm schedules only a $1/\log n$ fraction of the calls scheduled by the optimal algorithm.

If a limited delay is allowed, intuitively we should be able to improve the utilization. Surprisingly, we show that for any constant $c \geq 0$, the competitive ratio for both performance measures is still at least $\Omega(\log n)$, even on a linear array. This is true even if the on-line schedule is allowed to delay but is compared to an off-line schedule that is not allowed to delay requests at all. This implies that delaying the calls for a limited time does not help to improve the performance of call admission algorithms at all with respect to both data-admission ratio and call-admission ratio. Thus our lower bound generalizes the results of [ABFR94, LT94] in a very strong way.

Theorem 2.1 *Let $c \geq 0$ be a constant. Suppose R is a randomized call admission algorithm for a linear array network with $n+1$ nodes for requests with unit bandwidth requirements such that each request is either scheduled between T and $T + cd$, where T is its arrival time and d is its duration, or not scheduled at all. Then*

- *the competitive ratio of R with respect to data-admission ratio is at least $\Omega(\log n)$,*

- *the competitive ratio of R with respect to call-admission ratio is at least $\Omega(\log n)$,*

even if the off-line schedule has to satisfy every request immediately or reject it.

Proof. We assume that c is an even integer without loss of generality (we can always round it up). We denote the nodes of the linear array by $v_0, \ldots, v_n$. The notation (T, u, v, b, d) denotes a request that arrives at time T for a connection between nodes u and v with bandwidth requirement b and duration d.

We first construct a large set of requests and point out its important properties; later we choose a random subset of the large set of requests (according to a specific distribution) that we use as an actual input for an call admission algorithm.

Let $\bar{c} = 3c/2 + 1$, $\hat{c} = 2\bar{c} + 1 = 3c + 3$, and $l = \lfloor \log_{\hat{c}} n \rfloor$.

The requests are indexed by sequences $\alpha = (\alpha_1, \ldots, \alpha_{|\alpha|})$ of odd integers from $[1, \hat{c} - 1]$ of length $|\alpha| \leq l$. The request indexed by α arrives at time $\mathrm{Arr}(\alpha)$ which is defined as $(\alpha_1 \hat{c}^{l-1} + \alpha_2 \hat{c}^{l-2} + \ldots + \alpha_{|\alpha|} \hat{c}^{l-|\alpha|})(c+1)$, i.e., as if we fill the sequence by zeros and interpret it as a number with l digits written in the basis $\hat{c}$, which is then multiplied by $c + 1$. Let the request indexed by α be $r_\alpha = (\mathrm{Arr}(\alpha), v_0, v_{\hat{c}|\alpha|}, 1, \hat{c}^{l-|\alpha|})$, see Figure 1. Note that the time interval during which the request r_α may be running starts at $\mathrm{Arr}(\alpha)$ and is $(c+1)\hat{c}^{l-|\alpha|}$ long.

Since the amount of data of each request is the same, the data-admission ratio is equal to the call-admission ratio. Since all requests require full bandwidth on the edge between v_0 and v_1, the time intervals in which the satisfied requests are scheduled have to be pairwise disjoint.

The requests are naturally arranged in a $\bar{c}$-ary tree, with the smallest element $r_\emptyset$ at the root, and $r_{\alpha'}$ being the descendant of r_α if α is a subsequence of α'. The following are the only two properties of intervals we need for the rest of the proof; both are easy to check.

(1) Whenever the on-line algorithm schedules (within the appropriate interval) a request r that is not on the last level of the tree, there is a child r' of r in the tree such that r' did not arrive yet and no request in the subtree of r' can be scheduled (without a collision with r).

(2) If the off-line algorithm schedules (with no delay) a request r_α, it does not collide with any request in the tree below r except for the requests in the tree below $r_{\alpha 1}$.

Now we define the random subset $\mathbf{R}$ of requests that we use as an input; in fact it is a subtree of the tree defined above. It is defined recursively as follows:

- $r_\emptyset \in \mathbf{R}_l$;
- for any $r_\alpha \in \mathbf{R}_l$, $|\alpha| < l$ and i an odd number from $[1, \hat{c} - 1]$, we put $r_{\alpha i}$ in $\mathbf{R}_l$ with probability $1/\bar{c}$.

All probabilistic choices are made independently. In other words, the set of requests $\mathbf{R}_{l+1}$ contains a root of the tree and then independently for each of $\bar{c}$ possible childs with a probability $1/\bar{c}$ we put in a random subtree from $\mathbf{R}_l$. By $\#(\mathbf{R}_l)$ we denote the number of children of the root. From the definition of $\mathbf{R}_l$ it follows that the expected number of children $\mathrm{E}[\#(\mathbf{R}_l)]$ is 1 for $l > 0$. Recursively, the expected number of the request of the same level in the tree that

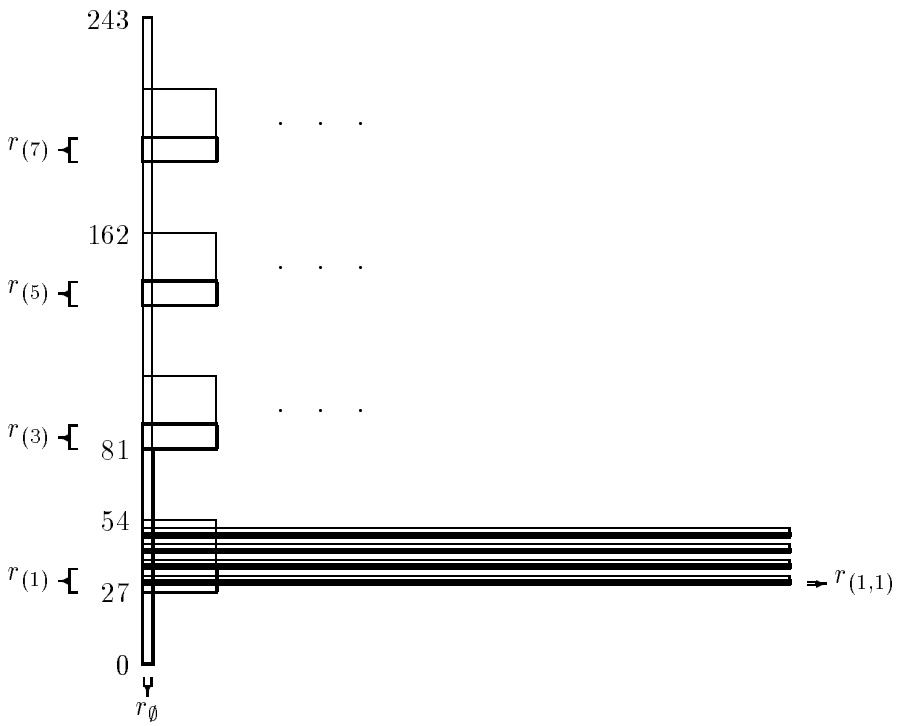

Figure 1: An example of the large set of requests for $n = 243$ and $c = 2$. Horizontal dimension denotes the nodes in the linear array, vertical dimension is the time. The thick rectangles denote the requests as if they were scheduled immediately after their arrival, the thin rectangles delimits the area within which each request has to be scheduled.

are in $\mathbf{R}_l$ is 1 for any level. Thus $\mathbf{R}_l$ behaves very much as a random branch of the tree, but the independence makes our considerations much easier.

First we compute the expected number of requests that the optimal schedule can satisfy. We denote this number $\mathrm{OPT}(l)$ and compute recursively in l. For $l = 0$, $\mathrm{OPT}(l) = 1$, since the optimal schedule will schedule the single request. Now we compute $\mathrm{OPT}(l+1)$ as a function of $\mathrm{OPT}(l)$. The optimal strategy optimally satisfies the requests of the subtrees rooted at all the existing children, and then also schedules the root request if possible, which is if and only if the first child is not present (recall the condition (2)). Thus the expected number of satisfied requests is $\mathrm{E}[\#(\mathbf{R}_{l+1})]\mathrm{OPT}(l) + (1 - 1/\bar{c}) = \mathrm{OPT}(l) + (1 - 1/\bar{c})$, and hence $\mathrm{OPT}(l) = \Theta(l)$.

Now we bound the maximal (expected) number $F(l)$ of requests satisfied by any on-line algorithm. We prove by induction on l that $F(l) \leq \bar{c}$. For $l = 1$, clearly $F(1) = 1 \leq \bar{c}$. For $l+1$ we distinguish two cases. First, suppose that on some partially known input the algorithm decides to schedule the root request. Then (according to the condition (1)) the whole subtree of one of the possible children is blocked; since it is before the arrival time of those requests, the on-line algorithm has no possibility of knowing if they are in the random set $\mathbf{R}_{l+1}$. Thus, averaging over the instances consistent with this partial input, we get that there is on average at least $1/\bar{c}$ child tree that is blocked by the root request and hence the number of satisfied requests is at most $1 + (\mathrm{E}[\#(\mathbf{R}_{l+1})|\text{partial input}] - 1/\bar{c})F(l) \leq \bar{c}\mathrm{E}[\#(\mathbf{R}_{l+1})|\text{partial input}]$, using the induction assumption $F(l) \leq \bar{c}$. Second, if for some instance R with non-zero probability in $\mathbf{R}_{l+1}$ the on-line algorithm does not schedule the root job, then the number of satisfied requests is bounded by $\#(R)F(l) \leq \bar{c}\#(R)$. Averaging over all instances we get that the average number of satisfied requests $F(l+1)$ is bounded by $\bar{c}\mathrm{E}[\#(\mathbf{R}_{l+1})] = \bar{c}$. Note that the fact that the algorithm can be randomized does not change the analysis above.

Suppose that we have a randomized σ-competitive algorithm. By averaging over all instances from our distribution and the definition of competitiveness, it has to satisfy at least $\mathrm{OPT}(l)/\sigma$ requests. Since by the above analysis it can satisfy only a constant number, $\bar{c}$, of the requests, the competitive ratio is at least $\Omega(\mathrm{OPT}(l)) = \Theta(\log n)$. □

3 The GREEDY algorithm

For the rest of this paper there is no limit on the delay allowed before a call is scheduled and the duration of a request is unknown to the on-line algorithms until the connection is terminated. We also assume that all requests are available at time $t = 0$, i.e., our algorithms are batch-style. All our algorithms can be converted to fully on-line algorithms at a cost of either increasing the competitive ratio by 1 or increasing the competitive ratio by a factor of 2, as shown in Section 6. We also assume that time can be broken into time steps, and require that each duration be an integral number of time steps.

Our first algorithm, GREEDY, schedules each request r at the first time t for which enough bandwidth is available for r. In this section we analyze its performance with respect to makespan on simple networks such as trees and networks with small separators.

3.1 The GREEDY algorithm and its basic property

3.1.1 GREEDY for tree networks

A formal description of the GREEDY algorithm for tree networks is presented below. The tuple (u, v, b, d) denotes a request for a connection between nodes u and v requiring bandwidth b and duration d (d is not given in the input).

Algorithm GREEDY for trees

Given a batch of requests $r_0, r_1, \ldots, r_m$ do:

$t := 0$;

repeat until all requests have been scheduled

for $i := 0$ **to** m **do**

let $r = r_i = (u, v, b, d)$ be the i-th request in the batch;

if r has not yet been scheduled **and** bandwidth $b' \geq b$ is available along a path (u, v) from u to v

schedule r along path (u, v) at time t;

$t := t + 1$;

In Section 3.2 we show that GREEDY is $O(\log n)$ competitive with respect to makespan for tree networks if the bandwidth requirements are either not too large or not too small. In case neither of these restriction is acceptable we present an $O(\log n)$-competitive batch-style algorithm $\overline{\text{GREEDY}}$ without restrictions on the bandwidth requirements. The cost of removing this restriction is that we need to partition the requests into two sets and process each set separately. This partitioning is unavoidable because GREEDY may perform arbitrarily badly (Section 3.3) if arbitrary bandwidth requirements are allowed. In light of the examples of Section 3.3, it is somewhat surprising that GREEDY is $O(\log n)$-competitive even if arbitrarily small (but not arbitrarily large) values for the bandwidth requirements are allowed.

Algorithm $\overline{\text{GREEDY}}$

Given a batch of requests $R = \{r_0, r_1, \ldots, r_m\}$ do:

Partition the requests into two classes R_1 and $R_2 = R - R_1$, so that R_1 contains exactly all the requests with bandwidth requirements less than $1/2$;

(1) Schedule batch R_1 using GREEDY;

(2) Schedule batch R_2 using GREEDY;

3.1.2 GREEDY for networks with small separators

In order for GREEDY to achieve similar performance bounds on any network with small edge separator trees as on tree networks we need to restrict the possible set of paths GREEDY is allowed to use. This restriction is necessary because of the following example. Given a 1-dimensional torus network of n nodes with edge capacities 1 and a list of requests $L = r_0 =$

$(0, 1, 2/3, 1); r_1 = (1, 2, 2/3, 1), \ldots, r_{n-1}(n-2, n-1, 2/3, 1)$. Imagine that GREEDY would route r_0 the long way. Then no other request can be scheduled during time step 0. If GREEDY does the same with all other requests the length of the schedule will be $n-1$ time steps long while the optimal schedule has length 1.

We start the discussion on the restriction of the set of paths by defining edge separator trees. Recall that a graph (V', E') is a vertex-induced subgraph of (V, E) if $V' \subset V$ and $E' = E \cap (V' \times V')$.

Definition 3.1 *A set of edges S* separates *a graph (V, E) into graphs (V_1, E_1), ..., (V_a, E_a) if every (V_b, E_b), $1 \leq b \leq a$, is a connected vertex-induced subgraph of (V, E) and E is a disjoint union of S, E_1, ..., E_a.*

An edge separator tree *ST for a graph G is a rooted tree in which each vertex w is labeled by a pair (G_w, S_w), so that for every vertex w and its children $w_1, \ldots, w_a$ in ST, S_w separates G_w into $G_{w_1}, \ldots, G_{w_a}$; moreover, if w is the root of ST then $G_w = G$.*

Let the size *of the separator tree ST, denoted $size(ST)$, be the maximum sum of sizes (number of edges) of the separators S_w along a path from the root to a leaf in a separator tree ST.*

Note that it follows from the definition that every G_w in ST is a connected vertex-induced subgraph of G, and for each leaf of ST w, S_w contains all edges of G_w.

The algorithm GREEDY for networks with edge separator tree ST is allowed to route requests only along reasonable paths with respect to ST, which are defined below.

Definition 3.2 *For a request $r = (u, v, b, d)$, let $z_r \in ST$ be the (unique) vertex such that both u and v are in G_{z_r}, but for each child w of z_r, G_w contains at most one of u and v. A path from u to v is* reasonable *with respect to ST if it is contained in G_{z_r}.*

It is easy to check that for any separator tree and any request there exist at least one reasonable path connecting them (the fact that graphs G_w are connected is needed here). Note that the length of a reasonable path and the $size(ST)$ of the separator tree ST are not necessarily related and that the unique path connecting u and v in a tree is always a reasonable path. The next lemma gives the important property of reasonable paths.

Lemma 3.3 *If reasonable paths for two requests r and r' (with respect to the same separator tree ST) intersect then either z_r is a descendant of $z_{r'}$ in ST, or $z_{r'}$ is a descendant of z_r or $z_r = z_{r'}$.*

Proof. From the definition of a separator tree it follows that otherwise the graphs G_{z_r} and $G_{z_{r'}}$ are disjoint, and so are the reasonable paths, as they are contained in these graphs; this contradicts the assumption. □

Now we can give a formal description of GREEDY on networks with small edge separators. We will show later that GREEDY on tree networks is just a special case of this algorithm.

Algorithm GREEDY for networks with edge separator tree ST

Given a batch of requests $r_0, r_1, \ldots, r_m$ do:

$t := 0$;

repeat until all requests have been scheduled

  for $i := 0$ **to** m **do**

    let $r = r_i = (u, v, b, d)$ be the i-th request in the batch;

    if r has not yet been scheduled

      and bandwidth $b' \geq b$ is available along a reasonable path p from u to v

    schedule r along path p at time t;

  $t := t + 1$;

The GREEDY algorithm cannot delay a request without a good excuse, meaning there has to be a set of requests scheduled instead of the delayed one such that all of these requests conflict with the delayed request and the sum of their bandwidth requirements is sufficient to prohibit the delayed request from being scheduled. This GREEDY property is stated more formally in Lemma 3.4.

Lemma 3.4 (GREEDY property) *If request $r = (u, v, b, d)$ is scheduled at time t then for every reasonable path p for r and for every time $t' < t$ there exists an edge e with capacity $c(e)$ on the path p such that the total bandwidth of all the requests using e at time t' exceeds $c(e) - b$.*

Proof. Assume that this does not hold for a path p and a time step $t' < t$. Then GREEDY would have scheduled r using path p during this time step, contrary to the assumption. □

3.2 Analysis of GREEDY

In this section we show that GREEDY is $O(size(ST))$-competitive with respect to makespan on n-node network with an edge separator tree ST provided that there is some constant c such that the bandwidth requirement of each request is at most $1-1/c$. In practice this is hardly a restriction at all, since the price of a connection requiring almost the whole bandwidth will be huge and only a few applications need such a large percentage of the link bandwidth (cf. [AW92]). In addition we show that GREEDY is $O(size(ST))$-competitive with respect to makespan if all bandwidth requirements are relatively large and that $\overline{\text{GREEDY}}$ is an $O(size(ST))$-competitive batch-style algorithm without restrictions on the bandwidth requirements. The results for GREEDY on tree networks are a special instance of these results.

Theorem 3.5 *Given a constant c, a network G of n nodes with unit edge capacities, and an edge separator tree ST for G, then*

(i) GREEDY is $(c+1)size(ST)$-competitive with respect to the makespan if the bandwidth requirement of each request is at most $1 - 1/c$.

(ii) GREEDY is $(c+1)size(ST)$-competitive with respect to the makespan if the bandwidth requirement of each request is at least $1/c$.

(iii) $\overline{GREEDY}$ is $(6\ size(ST))$-competitive with respect to the makespan.

Proof. (i) and (ii): Before proceeding with the proof, we first introduce some notation.

Let t_{OPT} be the makespan of an optimal schedule and let t_{GREEDY} be the makespan of the schedule generated by GREEDY. Let d_{max} denote the maximum duration of any request.

Suppose $w_0, w_1, \ldots, w_k$ is a path in the separator tree ST starting at the root. For any $i \leq k$, let R_i be the set of all requests r such that $z_r = w_i$ (i.e., the endpoints of r are both in G_{w_i} but not in $G_{w'}$ for any child w' of w_i). Let $R = \bigcup_{i=0}^{k} R_i$. For this set of requests R, let $den(R)$ denote their density, which is bandwidth times duration summed over all of the requests in R, and let $den_t(R)$ denote the density at time t, which is the sum of the bandwidth of all requests from R running at time t in the schedule generated by GREEDY. Obviously $den(R) = \sum_{t=1}^{t_{GREEDY}} den_t(R)$.

First we construct a path $w_0, w_1, \ldots, w_k$, such that for all but $(k+1) * d_{max}$ time steps t, $den_t(R) \geq 1/c$. We construct the path from the root w_0 down. In order to determine which child of vertex w_i on the path will be w_{i+1}, we check the children w'_b of w_i in ST to see if there is some request with both endpoints in the corresponding subgraph $G_{w'_b}$. If no subgraph has such a request, the path terminates at w_i. Otherwise, if only one child's subgraph has such a request, then that child becomes w_{i+1}. If several children's subgraphs have such requests, then we choose the child whose subgraph contains a request with the latest finishing time in the schedule (breaking ties arbitrarily).

We define r_i to be the request scheduled by GREEDY on a path contained in G_{w_i} that finishes latest of all such requests (ties broken arbitrarily). Note that r_i's are not necessarily distinct. Let t_i be the time when r_i finishes in the schedule and b_i the bandwidth requirement of r_i. Since $G_{w_{i+1}}$ is a subgraph of G_{w_i} and $G_{w_0} = G$, we have $t_{i+1} \leq t_i \leq t_0 = t_{GREEDY}$.

First consider a time step t such that some r_i is running at time t. For such a time step $den_t(R)$ might be small (i.e., less than $1/c$). However, since there are at most $k+1$ requests, r_0 through r_k, and each has duration at most d_{max}, there are at most $(k+1) * d_{max}$ such time steps.

Second, suppose that no request r_i is running at time t. In this case we prove that $den_t(R) \geq 1/c$. Let $l \leq k$ be the largest number such that $t < t_l$ (since $t_0 = t_{GREEDY}$, such an l exists). It follows that the request r_l was not yet scheduled at time t. The GREEDY property (Lemma 3.4) guarantees that there exists a set of requests R' running at time t with density $den_t(R') > 1 - b_l$ all using an edge e on the path used by r_l. From the assumption about the bandwidths it follows that $den_t(R') \geq 1/c$: in the case (i) of the theorem it follows since $b_l \leq 1 - 1/c$; in case (ii) it follows since R' is nonempty and the bandwidth of any request is at least $1/c$. Now we prove that $R' \subseteq R$. Fix a request $r \in R'$ and look at the vertex z_r in ST. By the choice of l and the construction of w_{l+1} (or the fact that $k = l$), z_r is not a descendant of w_l. Since the paths used by r_l and r are reasonable, it follows by Lemma 3.3 that z_r is on the path from the root to w_l, and thus $z_r = w_{l'}$ for some $l' \leq l$. It follows that $r \in R$ and hence $R' \subseteq R$ and $den_t(R) \geq den_t(R') \geq 1/c$.

We have proved that for all but $(k+1) * d_{max}$ time steps t, $den_t(R) \geq 1/c$. Therefore, summing over all t,

$$den(R) \quad \geq \quad (t_{GREEDY} - (k+1) * d_{max})/c,$$

and

$$t_{GREEDY} \quad \leq \quad c * den(R) + (k+1) * d_{max}.$$

In the optimal schedule, any request from R_i must be scheduled using an edge from a separator $S_{w_{i'}}$ for some $i' \leq i$, Therefore for each of the $den(R)$ bits of data there exist some separator S_{w_j}, $j \leq k$ which this bit has to cross in time at most t_{OPT}. Since the total capacity of all these separators is at most $size(ST)$, we have $den(R) \leq t_{OPT} * size(ST)$. Thus

$$\begin{aligned} t_{GREEDY} &\leq c * den(R) + (k+1) * d_{max} \\ &\leq c * t_{OPT} * size(ST) + (k+1) * d_{max} \quad \leq \quad (c+1) size(ST) * t_{OPT} \end{aligned}$$

since $k+1 \leq size(ST)$ and $d_{max} \leq t_{OPT}$. Therefore GREEDY is $(c+1)size(ST)$-competitive. □

(iii) Since $\overline{\text{GREEDY}}$ uses the result of (i) with $c = 2$ to schedule requests of batch R_1 and the result of (ii) with $c = 2$ it follows that $\overline{\text{GREEDY}}$ is $(6 \ \ size(ST))$-competitive. □

For every n-node tree there is a node v whose removal separates the tree into components such that each contains at most $n/2$ nodes of the tree. Since in our case the tree has maximum degree 3 and there are at most 3 edges adjacent to v. The removal of the two edges leading to the two larger components separates the tree into at most three components, each containing at most $n/2$ nodes of the tree. Using this fact recursively, we can construct a separator tree ST with depth $\log_2 n$ and a size of each separator 2; hence $size(ST) = 2\log_2 n$. Corollary 3.6 now follows by observing that the unique path connecting u and v in the tree is always reasonable.

Corollary 3.6 *Given a constant c and an n-node tree network G with maximum degree 3 (e.g. any binary tree) and unit edge capacities then*

(i) GREEDY is $2(c+1)(\log_2 n)$-competitive with respect to makespan if the bandwidth requirement of each request is at most $1 - 1/c$.

(ii) GREEDY is $2(c+1)(\log_2 n)$-competitive with respect to makespan if the bandwidth requirement of each request is at least $1/c$.

(iii) $\overline{GREEDY}$ is $(12\log_2 n)$-competitive with respect to makespan.

We can use the above theorem on any planar network with bounded node degree, which includes e.g. 2-dimensional mesh networks. Lipton and Tarjan [LT79] showed that any planar graph has a vertex separator of size $O(\sqrt{n})$ such that the size of each component is a fixed fraction of the size of the graph. If the degree is bounded, it follows that we have an edge separator with the same parameters as well. If we apply this recursively, we get a separator tree ST with $size(ST) = O(\sqrt{n})$.

Corollary 3.7 *Given a constant c and any planar network with bounded degree of nodes and unit edge capacities then*

(i) GREEDY is $O(c\sqrt{n})$-competitive with respect to makespan if the bandwidth requirement of each request is at most $1 - 1/c$.

(ii) GREEDY is $O(c\sqrt{n})$-competitive with respect to makespan if the bandwidth requirement of each request is at least $1/c$.

(iii) $\overline{GREEDY}$ is $O(c\sqrt{n})$-competitive with respect to makespan.

3.3 Lower bounds on GREEDY's performance

In this section, we present two simple examples that show that

1. the competitive ratio of GREEDY, with respect to makespan, can be arbitrarily poor if the bandwidth of the requests are allowed to vary arbitrarily between 0 and 1 on a linear array network with unit capacity edges, and

2. the competitive ratio of GREEDY, with respect to makespan, is $\Omega(\log n)$ on an n-node tree, even if all edge capacities, bandwidth requirements, and durations are 1.

The first example, Figure 2, shows that if the bandwidth requirements are not restricted a request requiring only a little bandwidth can block a request that requires almost all of the bandwidth. This can, as in this example, result in bad schedule proving that there is a lower bound of $n/2$ on the competitive ratio of GREEDY for a linear array network of n nodes.

The second example, Figure 3, proves that GREEDY cannot be better than $(1/2)\log n$-competitive on n-node binary trees. In contrast (see Section 5.1), GREEDY is constant competitive on linear arrays with unit capacity edges, when all requests have unit bandwidth (and arbitrary durations).

4 Lower bound for call admission on a tree network

In this section we prove that no deterministic on-line call admission algorithm on a tree of $n = 2^{d+1} - 1$ nodes can achieve a competitive ratio better than $\frac{\log d}{\log \log d}$ with respect to makespan.

In Section 3.3, we give an example that proves that GREEDY cannot be better than $(1/2)\log n$-competitive with respect to makespan on n-node binary trees. For the lower bound of this section, the adversary will try to restrict the possibilities of the call admission algorithm so that the algorithm must act similarly to GREEDY in the example. The main difference between the two lower bounds is that the adversary can force GREEDY to schedule a specific subset of the requests, but it does not seem possible to force any on-line algorithm to do so. Thus, this lower bound is exponentially weaker.

The adversary proceeds by either setting the durations of a small number of jobs so that the on-line algorithm makes little progress during some period of time, or by finding an unused

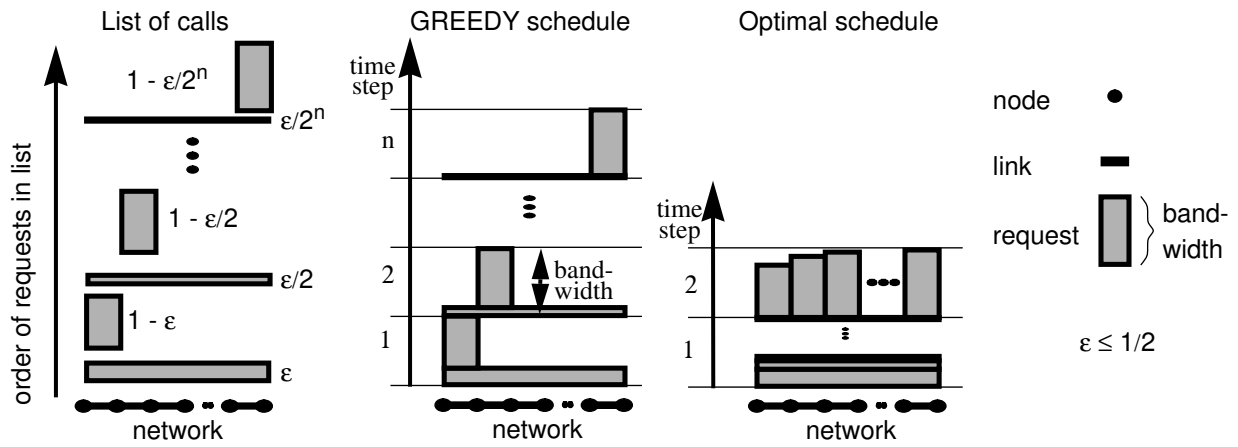

Figure 2: An example for a GREEDY schedule vs. an optimal schedule for an $(n+1)$-node linear array network. All requests arrive at time 0 and have unit duration, but their bandwidth requirements vary. In this figure the height of each request is proportional to its bandwidth requirement. Some of the requests span just a single edge of the network, while others span the entire network. In this figure width of a request is proportional to its span. The first picture shows the requests in the order they appear in the list of the GREEDY algorithm. The second picture shows the resulting GREEDY schedule while the third one gives the optimal schedule. In this example $\epsilon \leq 1/2$.

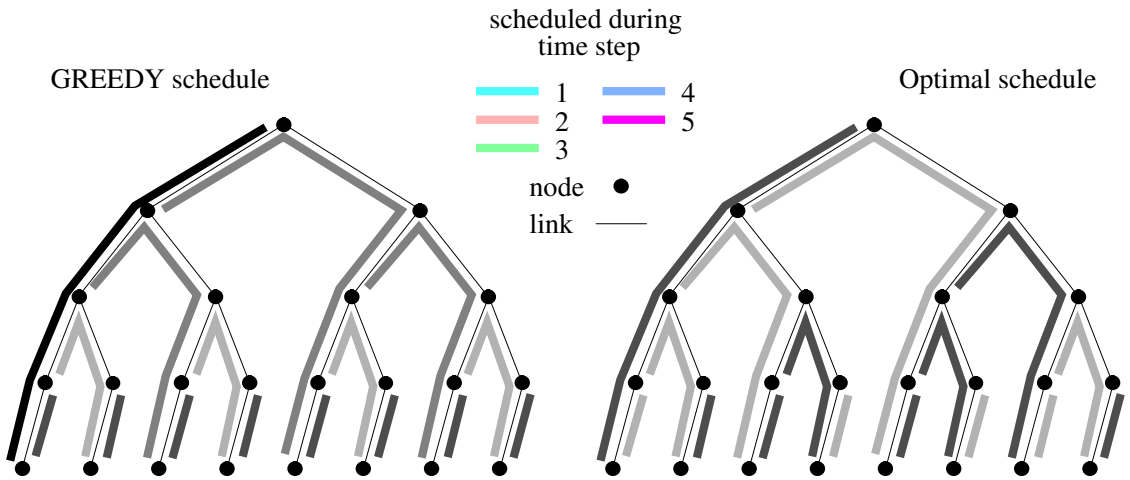

Figure 3: An example for a GREEDY schedule vs. an optimal schedule for a tree network. All capacities, bandwidth requirements, and durations are 1, and all requests arrive at time 0. The first picture shows the schedule produced by the GREEDY algorithm if the requests appear in the input list in the following order: if a request was scheduled at time t than it appears before all requests which were scheduled at a time greater than t. The second picture shows an optimal schedule for this set of requests.

part of the network. Simultaneously, the adversary finds a subtree of the network and a subset

of unscheduled requests that constitute a smaller size problem of the same kind.

Theorem 4.1 *Consider a network that is a full binary tree T of depth d with $n = 2^{d+1} - 1$ nodes and unit edge capacities. Suppose we have a deterministic call admission algorithm for requests with unit bandwidth requirements on such a network. Then the competitive ratio with respect to makespan is at least $\Omega(\frac{\log\log n}{\log\log\log n}) = \Omega(\frac{\log d}{\log\log d})$.*

Proof. The proof has the following outline. We first construct a set of requests which we use as input to the call admission algorithm. Second we describe the strategy the adversary will use to determine the duration of the requests. Next we provide a lower bound on the time the on-line algorithm needs to schedule this set of requests and finish by providing a better solution and thus bounding the competitive ratio.

The requests

As a naming convention for the nodes of the network each node $v = v_\alpha$ is indexed by a bit string α. The length of the string $|\alpha|$ is the depth of the node in the tree and its left and right sons are $v_{\alpha 0}$ and $v_{\alpha 1}$. A node v_α is a predecessor of node v_β if α is a initial segment of β, denoted $\alpha \prec \beta$. The root of the tree is v_ϵ. Let T_α denote the subtree of tree T rooted at node v_α.

All requests have unit bandwidth requirements and the source and destination nodes of the set of requests R are defined as follows:

$$R = \{(v_{\alpha 0}, v_\beta) \mid \alpha 1 \prec \beta \wedge |\beta| = n\}$$

The durations of the requests are determined dynamically by the adversary depending on the actions of the on-line call admission algorithm.

Adversary strategy

The *adversary* proceeds in phases. The goal of each phase is to set the durations of a small number of jobs so that the on-line algorithm makes little progress for some period of time, while it is possible to simultaneously find a subtree of the network and a subset of the unscheduled requests which constitute a smaller size problem of the same kind. After the adversary has performed l phases, the subtree is given by its root α_l and depth d_l (initially $\alpha_0 = \epsilon$, and $d_0 = d$). We define R_l to be the set of all requests whose durations were not specified during the first l phases; it always only contains requests within T_{α_l}, i.e. such that both the source and the destination are in T_{α_l}; in addition the source is never more than d_l levels below v_{α_l} (i.e. it is in the interesting subtree).

In each phase the adversary gives the algorithm one time step to schedule requests. After that he focuses on the set of nodes S_l which are at the level $d_l / \ln d$ of the tree T_{α_l}, with the intention to later use either the subtree above S_l or one of the subtrees rooted at some $\gamma \in S_l$. If for some such γ the algorithm schedules no request below γ, the adversary, in the NO REQUEST phase, removes all requests which do not lie completely within this subtree and continues with it. Note, the algorithm did not do any work in this subtree within the previous

time step, which ensures that the algorithm has wasted time. If the algorithm does schedule at least one request below each such γ, the adversary, in the BLOCKING phase, sets the duration of these requests so that the on-line algorithm makes no progress during a time period of length $t = \frac{\ln d}{2 \ln \ln d}$, and then continues with the subtree between α_l and S_l. Note that the depth of the subtree decreases in t NO REQUEST steps about as much as in 1 BLOCKING step, which is proportional to the duration of the steps.

The invariant which the adversary preserves throughout the schedule says that (1) for any two nodes in the interesting part of the tree there still exists a request which has to use both of them, if there was one at the beginning, and (2) all the remaining requests start in the interesting part of the tree. More formally,

1. For each pair of nodes $v_\gamma, v_\beta \in T_{\alpha_l}$ such that $\gamma 1 \prec \beta$ and $|\beta| = |\alpha_l| + d_l$, there exists $\beta' \succ \beta$, $|\beta'| = d$, such that $(v_{\gamma 0}, v_{\beta'}) \in R_l$.

2. For every $(v_{\gamma 0}, v_\beta) \in R_l$: $\gamma \in T_{\alpha_l}$ and $|\gamma| < |\alpha_l| + d_l$.

Adversary Strategy

$t := \frac{\ln d}{2 \ln \ln d}$;
$l := 0$; $\alpha_l := \epsilon$; $d_l := d$;
repeat until at least t_2 time steps have elapsed
let the on-line algorithm schedule requests for one time step
$S_l := \{\beta \in T_{\alpha_l} \mid |\beta| = |\alpha_l| + \lceil d_l / \ln d \rceil\}$;
if there exist a $\beta \in S$ such that in this time step no request from R_l was scheduled within T_β
NO REQUEST:
Choose a β and $a \in \{0, 1\}$ such that no request from R_l scheduled in this time step intersects $T_{\beta a}$
$\alpha_{l+1} := \beta a$; $d_{l+1} := d_l - \lceil d_l / \ln d \rceil - 1$;
Remove all scheduled requests;
Assign duration 0 to all requests that do not lie within $T_{\alpha_{l+1}}$;
else
BLOCKING:
$\alpha_{l+1} := \alpha_l$; $d_{l+1} := \lceil d_l / \ln d \rceil$;
for every $\beta \in S_l$,
Choose $r_\beta \in R_l$ to be one request scheduled in this time step within T_β;
Assign a duration to r_β so that it will finish t time steps from the beginning of this phase
Remove all other scheduled requests (i.e., except $\{r_\beta \mid \beta \in S_l\}$);
Assign duration 0 to all requests which do not use an edge of $\{(v_{\gamma 0}, v_\gamma) \mid (\exists \beta, \beta') : r_\beta = (v_{\gamma 0}, v_{\beta'})\}$ and to all requests that use no vertex of depth less than $|\alpha_{l+1}| + d_{l+1}$;

$l := l + 1$;

end repeat

Assign duration 0 to all remaining requests;

Remove all currently running requests;

Correctness of the Adversary Strategy

In this section we analyze the adversary strategy from the previous section. Throughout the analysis we assume that d is sufficiently large, and in particular $t > 1$.

First, note that in the BLOCKING phase it is always possible to choose a as required. Any request that intersects T_β has to use an edge from β to its parent since no request is scheduled within T_β, by the choice of β. There can be only one such request, and it can intersect only one of $T_{\beta 0}$ and $T_{\beta 1}$.

Second, the invariant stated above is preserved by the adversary strategy. It is obviously true at the beginning of the schedule and preserved during each NO REQUEST phase. Let us suppose that the invariant was true before BLOCKING phase l. Invariant (2) is preserved trivially. Fix $\gamma, \beta \in T_{\alpha_{l+1}}$ such that $\gamma 1 \prec \beta$ and $|\beta| = |\alpha_{l+1}| + d_{l+1}$. From the definition of α_{l+1} and d_{l+1} it follows that $\beta \in S_l$. Let $r_\beta = (v_{\gamma' 0}, v_{\beta'})$ (i.e., the request under β scheduled during phase l with duration more than 1). By the invariant (2) for l, there is $\beta'' \succ \gamma' 0$ such that $|\beta''| = |\alpha_l| + d_l$. Using the invariant (1) for l on γ and β'', there exists a request with source $v_{\gamma 0}$ which uses the vertex $v_{\beta''}$ and hence also v_β; thus the invariant (1) is true for $l + 1$ as well.

Third, we need to show that $d_l > 0$ for at least t^2 time steps.

Lemma 4.2 *If the time spent in the first l phases of the adversary strategy is $t_l \leq t^2$, then $d_l \geq d(\ln d)^{-t_l/t} \geq 2 \ln d$.*

Proof. First we prove the second inequality. We have $d(\ln d)^{-t_l/t} \geq d(\ln d)^{-\ln d / 2 \ln \ln d} = d^{1/2} \geq 2 \ln d$ for sufficiently large d.

Now we prove the first inequality by induction on l. For $l = 0$ it holds trivially.

Suppose that the lth phase is BLOCKING. Then $d_{l+1} = \lceil d_l / \ln d \rceil \geq d_l / \ln d$. The induction step follows since the time spent in this phase is t.

Suppose that the lth phase is NO REQUEST. Then $d_{l+1} = d_l - \lceil d_l / \ln d \rceil - 1 \geq d_l - 2d_l / \ln d = d_l(1 - 2/\ln d)$ (using the induction assumption $d_l \geq 2 \ln d$). To finish the induction step, we need to prove that $1 - 2/\ln d \geq (\ln d)^{-1/t}$, since the time spent in this phase is 1. We have $(1 - 2/\ln d)^t \leq 1 - 2t/\ln d = 1 - 1/\ln \ln d > 1/\ln d$ for sufficiently large d. □

As a consequence of this lemma and the invariant, R_l is nonempty for all l, and the length of the schedule generated by the algorithm is at least t^2.

Off-line solution

For the proof of the lower bound we still need to prove that an off-line call admission algorithm for the tree network can schedule the requests in less time.

The next lemma shows that all the requests can be scheduled efficiently. It is easy to schedule the requests with duration 1. For the requests with longer duration we use the fact that their conflict graph is similar to the one used in the example showing that GREEDY is $\Omega(\log N)$ competitive for tree network.

Lemma 4.3

1. *All the requests scheduled during the NO REQUEST phases can be scheduled in time at most* 1.

2. *All the requests with duration at most* 1 *scheduled during one BLOCKING phase can be scheduled in time at most* 1.

3. *All the requests with duration more than* 1 *scheduled during the BLOCKING phases can be scheduled in time at most* $2t$.

Proof.
1. The path used by any request scheduled in a NO REQUEST phase l is disjoint with the subtree $T_{\alpha_{l+1}}$, while all the requests scheduled later are within $T_{\alpha_{l+1}}$. This in particular means that any two requests scheduled in different NO REQUEST phases use disjoint paths and can be scheduled in parallel. Since all the requests scheduled during one NO REQUEST phase can be scheduled in time 1 using the schedule produced by the on-line algorithm, the total time needed to schedule them in parallel is 1.
2. Just use for these requests the same schedule as the on-line algorithm during that BLOCKING phase.
3. Suppose that two requests $r = (v_{\gamma 0}, v_\beta)$ and $r' = (v_{\gamma' 0}, v_{\beta'})$ intersect and are scheduled during BLOCKING phases l and l' with duration more than 1. Obviously $l \neq l'$, since the requests of non-zero duration scheduled during the same phase are disjoint. By symmetry we can assume $l < l'$. It follows from the algorithm that $|\gamma| > |\gamma'|$ and the edge $(v_{\gamma 0}, v_\gamma)$ is used by r', since otherwise r would be assigned duration 0 in BLOCKING phase l. (r cannot intersect $r_{\gamma''} \neq r'$ for $\gamma'' \in S_l$, since then it would be disjoint from r'.) It follows that each edge is used by at most two such requests, and all these requests can be partitioned into two sets of requests such that each set uses disjoint edges. Since the duration of all requests in this class is at most t, we can schedule them in time $2t$. □

Since each BLOCKING phase takes time t, there are at most t BLOCKING phases. Thus the last lemma guarantees that the makespan of the optimal schedule is at most $3t+1$. Since the algorithm generated a schedule of length at least t^2, the competitive ratio is at least $t^2/(3t+1) = \Omega(t) = \Omega(\log d / \log\log d)$. □

5 Algorithms for linear array networks

If each request specifies not only its terminal nodes and bandwidth requirement, but also the path that its connection will take through the network, and all link capacities, bandwidth

requirements, and durations are 1, then the call admission problem is equivalent to coloring the nodes of a graph that we call the *conflict graph*. In the conflict graph there is a node for each request and an edge between two nodes if the corresponding requests share a link of the network. A coloring of the nodes of the conflict graph corresponds to a schedule of the requests. Each color corresponds to a time step and the total number of colors used corresponds to the makespan of the schedule. The simple correspondence between call admission algorithms and coloring algorithms is lost once we introduce arbitrary durations for requests. In this case, if a request has duration greater than one, it must be assigned a collection of consecutive colors.

We can nevertheless use the relationship between coloring and call admission to show that GREEDY is constant competitive on a linear array if each edge has unit capacity and each request requires unit bandwidth. The proof shows that the schedule generated by the GREEDY algorithm corresponds to a coloring produced by a particular *on-line coloring algorithm* even with calls of arbitrary duration. An on-line coloring algorithm is one that is given the nodes of a graph one-by-one according to a specific sequence. As each node is revealed, the edges connecting that node to previously seen nodes are also revealed. The on-line coloring algorithm must assign a color to each node immediately after seeing it, and once assigned, the color cannot be changed.

Given a linear array network the conflict graph is an *interval graph*. In an interval graph the nodes can be represented by nonempty intervals on the real axis, and an edge exists between each pair of nodes whose intervals intersect. In the next two sections we show how to apply results proven in the context of on-line coloring of interval graphs to the call admission problem for linear array networks.

5.1 GREEDY for linear array networks

The simplest on-line coloring algorithm assigns each node u the color α which is the least positive integer not already assigned to a neighbor of u. This algorithm is called *First-Fit*. For interval graphs, Kierstead [Kie88] showed that First-Fit is 40-competitive with respect to the number of different colors used, and later Kierstead and Qin [KQ92] improved this bound to 25.8. Chrobak and Slusarek [CS88] give an example that proves that First-Fit cannot be better than 4.4-competitive.

Theorem 5.1 *Given a linear array network with unit capacity edges, and requests with unit bandwidth requirements and unknown durations, GREEDY is* 25.8*-competitive with respect to the makespan and cannot be better than* 4.4*-competitive.*

Proof. The upper bound is shown via reduction to the on-line coloring problem of interval graphs, as described in Section 5.

For the reduction we show that for any given set of requests the schedule produced by GREEDY on a list of requests $r_0, r_1, \ldots, r_m$ and the coloring produced by First-Fit on some sequence L is the same. Given the GREEDY schedule, the input sequence for the coloring algorithm, L, is constructed in the following way:

Start at time step zero, $t = 0$. For all requests $r_i = (u_i, v_i, 1, d_i)$ running at time step t append node $[u_i, v_i]$ (of the interval graph) to the sequence L. Then increment t by 1.

In this construction L contains d_i copies of the node $[u_i, v_i]$ for each request r_i. The first copy has edges to all copies of all nodes $[u_j, v_j]$ that occur earlier in L such that the intervals $[u_i, v_i]$ and $[u_j, v_j]$ intersect. In addition, the j-th copy of $[u_i, v_i]$ has the same neighbors as the first, plus an edge to each of the first $j - 1$ copies. There are no other additional edges since all nodes corresponding to requests running at time t in the schedule produced by GREEDY are independent. Therefore all d_i copies generated for request r_i are colored by First-Fit with consecutive colors, which correspond to consecutive time steps.

Assume that r_i is the first request that GREEDY scheduled at time t but that First-Fit assigned a color $t' \neq t$. There are two cases to consider. First, suppose that $t' > t$. This is the easier case. Since GREEDY scheduled r_i at time t, and this is the first difference between First-Fit and GREEDY, First-Fit hasn't used color t yet. Thus, first-fit is obliged to use color t rather than any color $t' > t$. Now suppose that $t' < t$. The GREEDY property says that GREEDY always has to have an excuse for delaying a request. Therefore, GREEDY must have scheduled other requests that intersected r_i at each of steps 1 through $t - 1$. Therefore the neighborhood of $[u_i, v_i]$ has to include at least $t - 1$ edges to nodes occurring earlier in the sequence L. Since this is the first difference between GREEDY and First-Fit, these nodes must use all of the colors 1 through $t - 1$. Thus, First-Fit cannot use any color $t' < t$.

Therefore the schedule generated by GREEDY and the coloring of First-Fit for this sequence are equivalent. Since First-Fit, independent of the specific sequences, uses no more than 25.8 times the optimal number of colors, GREEDY is 25.8-competitive with respect to makespan.

The lower bound is shown by translating a difficult example from [CS88] for on-line coloring to the call admission problem. □

5.2 The STEADY algorithm for linear array networks

The results of Section 5.1 leave a large gap between the upper and lower bounds for the competitive ratio of GREEDY with respect to makespan for the linear array network. In this section, we describe a non-greedy algorithm called STEADY. We show that STEADY is 2-competitive with respect to makespan on a linear array with unit capacity edges, provided that all requests have bandwidth 1/2. We then extend this result to show STEADY is c-competitive if the bandwidth required by each request is between $1/c$ and $1/2$, $c \geq 2$.

In the call admission problem with arbitrary durations, an algorithm has to cope with the fact that at each time step some connections scheduled at an earlier time are still established. Therefore just finding independent sets in the conflict graph as for coloring is not sufficient. In each time step a set of already established requests needs to be augmented by additional requests such that all requests stay independent. The goal is to minimize the number of such steps. The algorithm STEADY is capable of doing this by scheduling requests of several independent sets

at the same time. STEADY makes steady progress in reducing workload and does not block any bandwidth unnecessarily.

We begin by giving an algorithm from the literature for on-line coloring of interval graphs. This algorithm is presented in [Slu89], where it is called `Algorithm SCC`.

We require one definition. The *density* of an interval I with respect to a set of intervals R, written $\textsc{Den}(I/R)$, is the maximum number of intervals of R that overlap I at a single point. That is,

$$\textsc{Den}(I/R) = \max_{x \in I} |\{r \in R | x \in r\}|$$

Algorithm SCC is presented with a sequence of intervals $I_1 \ldots I_n$ in order, and assigns each interval to a numbered *Bucket* $B_1, B_2, \ldots$, according to the *Bucket Rule*:

Bucket Rule: An interval I is assigned to the lowest numbered bucket B_i such that $\textsc{Den}(I/ \cup_{j \leq i} B_j) \leq i$.

In other words, an interval is placed into the first bucket i for which its total density in that and all preceding buckets is less than or equal to i. Slusarek proved in [Slu89] that if D is the maximum density of the interval graph no more than D buckets are used and that the density of each bucket is no more than two.

We now define algorithm STEADY by adapting the Bucket Rule to the call admission scenario. For each request r_i there is a natural interval (u_i, v_i) of a linear array. We abuse notation slightly below by using r_i to also refer to this interval. A request r of duration d can be thought of as d copies of the interval r. Each bucket is equivalent to a time step. STEADY is designed to find buckets according to increasing time steps such that the following invariants are maintained: copies of the same request are assigned to consecutive buckets and the density of each bucket is no more than two.

Algorithm STEADY

let R be the list of all requests $R = (r_0, \ldots, r_m)$;
let $t := 1$;
let $B_{t-1} := \{\}$;
repeat until all requests are scheduled
(1) $B_t := B_{t-1} - \{r | r$ finished$\}$;
  for all $r \in R$ **do**
  (2) **if** $\textsc{Den}(I/ \cup_{i \leq t} B_i) \leq t$ **then**
    (3) schedule r;
      let $B_t := B_t \cup r$;
      let $R := R - r$;
  $t := t + 1$;

Theorem 5.2 *If all requests have bandwidth between* $1/c$ *and* $1/2$, $c \geq 2$, *then STEADY is c-competitive*

Proof. Step (1) insures that copies of the same request are assigned to consecutive buckets such that no request once scheduled is terminated before it is completed. Let D be the maximum density of the interval graph corresponding to the set of requests R (including the copies for a request of duration greater than one). Step (2) ensures that no more than D buckets can be used. Assuming that the density of each bucket B_t is no more than two all requests assigned to one bucket can be scheduled at the same time since all bandwidth requirements are no more than $1/2$. This leaves us to show that Step 3 is only executed if the density of $B_t \cup r$ is less or equal to two. Assume that t is the first time that the density of B_t is greater than 2. Then Step 1 cannot have caused this problem and therefore Step 3 is at fault. This possibility is eliminated by the same arguments which are used in case of on-line coloring [Slu89] to prove that the density of each bucket is no more than two.

Let $R' \subset R$ be the requests which together establish the maximum density D. Than the optimal algorithm cannot schedule these requests in time less than D/c since they use the same edge and each of the requests needs bandwidth of at least $1/c$. The competitive ratio is therefore $\frac{D}{D/c} = c$. □

6 Conversion of batch-style algorithms to on-line algorithms

In this section we show that any algorithm for processing a single batch of requests that is k-competitive with respect to makespan can be converted into an algorithm that can process requests that arrive while others are being serviced that is $2k$-competitive with respect to makespan. We call the former algorithm a *batch-style* algorithm and the latter an *fully on-line* algorithm. This result follows (pretty much immediately) from a theorem of Shmoys, Wein, and Williamson [SWW91]. By placing some mild restrictions on the type of batch-style algorithm allowed, we strengthen this result by showing how to convert a k-competitive batch-style algorithm into a $(k+1)$-competitive fully on-line algorithm. An algorithm that satisfies these restrictions is called *mergeable*. Both the GREEDY algorithm of Section 3 and the STEADY algorithm of Section 5.2 are mergeable.

Given a k-competitive batch-style algorithm $\mathcal{A}$ for a scheduling problem, we create a $2k$-competitive fully on-line algorithm $\mathcal{A}'$ as follows: $\mathcal{A}'$ begins by scheduling all requests that have been presented by time 0 using $\mathcal{A}$. Once the schedule runs to completion, $\mathcal{A}$ is used again to schedule all new requests that have arrived between time 0 and the end of the first schedule. This process continues for as long as necessary.

Theorem 6.1 (SWW) *If $\mathcal{A}$ is a batch-style algorithm that is k-competitive with respect to makespan, then $\mathcal{A}'$ is an fully on-line algorithm that is $2k$-competitive with respect to makespan.*

This technique allows us to employ any batch-style algorithm as a black box to generate an fully on-line algorithm. In certain cases, however, a k-competitive batch-style algorithm may have an internal structure that allows us to convert it into an fully on-line algorithm that is

$(k+1)$-competitive. Algorithms that allow this more efficient conversion can incorporate new requests immediately into the schedule under construction. We refer to these algorithms as *mergeable* algorithms. More formally:

Definition 6.2 *Algorithm $\mathcal{A}'$ is a* mergeable variant *of $\mathcal{A}$ if*

1. *$\mathcal{A}'$ performs identically to $\mathcal{A}$ on batch-style request sequences (i.e., sequences in which all release times are 0).*

2. *Upon being presented with a set of new requests at time t_{new}, $\mathcal{A}'$ incorporates them into its schedule so that the set of requests running (or beginning) at t_{new} is identical to the set of requests that $\mathcal{A}$ would schedule to begin at time 0 if presented at time 0 with a batch $\mathcal{I}'$ of requests, where the sequence $\mathcal{I}'$ satisfies the following properties.*

   - *For each request r running at time t_{new} in the schedule produced by $\mathcal{A}'$, $\mathcal{I}'$ contains a request whose duration is $end(r) - t_{new}$, where $end(r)$ denotes the completion time of r.*
   - *$\mathcal{I}'$ also contains all requests that have not been scheduled to begin before time t_{new} in the schedule produced by $\mathcal{A}'$.*
   - *$\mathcal{I}'$ does not contain any requests that have completed by time t_{new} in the schedule produced by $\mathcal{A}'$.*

If $\mathcal{A}'$ exists then $\mathcal{A}$ is a mergeable *algorithm.*

The following theorem shows that mergeable algorithms can be converted to accept arbitrary release times with less overhead than general batch-style algorithms.

Theorem 6.3 *If $\mathcal{A}$ is a batch-style algorithm that is k-competitive with respect to makespan, and $\mathcal{A}'$ is a mergeable variant of $\mathcal{A}$ then $\mathcal{A}'$ is an fully on-line algorithm that is $(k+1)$-competitive with respect to makespan.*

Proof. Assume that algorithm $\mathcal{A}'$ when processing problem instance $\mathcal{I}$ is presented with the last request at time t_l. From this point on $\mathcal{A}'$ must be k-competitive on the requests that have not yet run to completion because $\mathcal{A}'$ is a mergeable variant of $\mathcal{A}$, and $\mathcal{A}$ is k-competitive on any batch. Since the optimal schedule for these requests has makespan no bigger than the optimal schedule for all requests, algorithm $\mathcal{A}'$ will complete no later than $t_l + k \cdot \mathrm{OPT}(\mathcal{I})$. But the optimal schedule must also take time at least t_l to run, since it cannot schedule the last request until then. So the total makespan of algorithm $\mathcal{A}'$ is bounded by $(k+1)\mathrm{OPT}(\mathcal{I})$.□

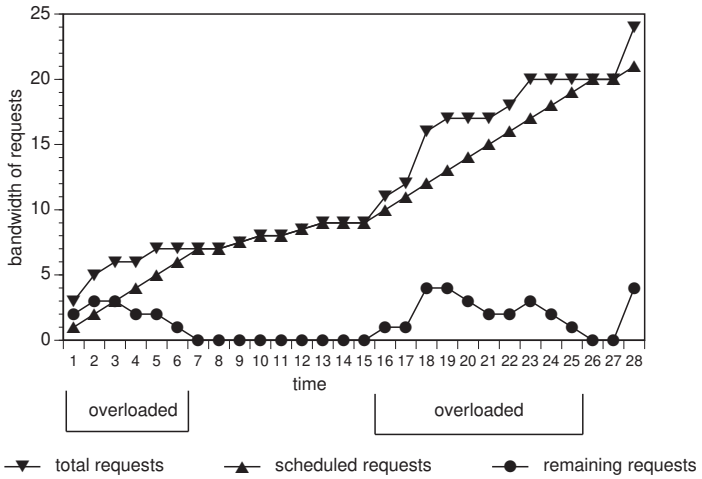

Figure 4: An example to visualize the concepts of over- and underload of a network

7 Bounding the maximum response time

As mentioned before, difficult situations usually do not arise as long as the network is lightly loaded. This section formalizes when a tree-like network is overloaded and when it is underloaded. For the purpose of this section, requests can arrive at any time and we assume unit capacity edges.

Intuitively, a network is overloaded at time t if the bandwidth required by the requests arriving at t is higher than the bandwidth available to satisfy these requests. It stays overloaded until it has scheduled all pending requests. Figure 4 gives an example based on the simplest possible network; one edge of capacity 1 and requests with duration 1. The total bandwidth of requests over time is plotted. At each time step some number of requests will arrive. The set of all requests arrived up to and including the current time step constitutes the *total requests* in Figure 4. In each time step the example network can only schedule requests with total bandwidth requirement of not more than 1. Therefore the *scheduled requests* can only grow by a value up to 1. If more requests arrive than can be scheduled during a time step (e.g. at time step 18) this network has to queue some of them. For each time step the bandwidth of the queued requests is shown as the *remaining requests*; it is the difference between the total requests and the scheduled requests. We consider a network to be overloaded until it has a chance to satisfy all requests. This is the case from time step 1 to 7, from time step 16 to 26, and from time step 28 on. Since the network is idle during time steps 8 and 27 and not fully utilized from time step 9 to 15, we consider it underloaded during these time steps.

One would like to consider a network overloaded if even an optimal schedule, (computed with unlimited computing power and full information) cannot schedule all requests immediately. For general networks it is difficult to reason about the optimal schedule. However, for networks with small edge separators a good lower bound for the performance of any optimal schedule of requests R up to a certain time t is the *density*, $\textsc{Den}(R, t)$. We defined a similar notion of density for intervals in Section 5.2. We now extend the definitions:

The density of an edge e at a time t, with respect to a set R of requests, written $\textsc{Den}_e(R,t)$, is defined to be the amount of communication from requests of R that has taken place across e through time t. Note that a request beginning before t but ending after t will contribute to the traffic based on the fraction of its entire duration that has completed by time t. More formally:

$$\textsc{Den}_e(R,t) = \sum_{r_i \in R | e \in r \& T_i \leq t} b_i \cdot \min(d_i, t - t_i)$$

We also define $\textsc{Den}(R,t) = \max_e \textsc{Den}_e(R,t)$.

We use the density to describe the workload of the network at any time t. In each time step all pending requests constitute the current workload. Even an optimal algorithm can reduce the workload by at most the maximum capacity in each time step. All new arriving requests increase the workload. So the workload at the next time step is the maximum of 0 and the workload of the previous time step minus the capacity plus the workload of the newly arriving requests.

This intuition is useful if the on-line algorithms could be as good as the optimal schedule. Since this is a rather unrealistic expectation we compare the performance of the algorithms against networks which have reduced network capacity. More precisely, we compare a k-competitive algorithm (with respect to the makespan) to a schedule which is allowed to reduce its workload by at most $1/k$ of the network capacity. The resulting workload corresponds to the workload of a network that can only use a $1/k = \alpha$ fraction of its capacity ($0 < \alpha \leq 1$). We call such a workload an α-workload. The following definition uses the set of requests $R = \{r_i = (T_i, u_i, v_i, b_i, d_i)\}$ and a set of requests R' derived from R as follows: if $r_i = (T_i, u_i, v_i, b_i, d_i) \in R$ then d_i requests r_{ij} between the same nodes as r_i just of duration 1 and arrival time $T_i + j$ are in R'; $r_{ij} = (T_i + j, u_i, v_i, b_i, 1)$.

Definition 7.1 *At time t the α-workload $wl^\alpha(t)$ of a tree network and the α-workload $wl^\alpha_e(t)$ of an edge e in such a network for a set of requests R is defined as:*

$$\begin{aligned}
R &= \{r_i | r_i = (T_i, u_i, v_i, b_i, d_i)\} \\
R' &= \{r_{ij} | r_{ij} = (T_{ij} = T_i + j, u_i, v_i, b_i, 1) \wedge r_i \in R \wedge 0 \leq j \leq d_i\} \\
wl^\alpha_e(t) &= \begin{cases} \textsc{Den}_e(\{r_{ij} | T_{ij} = 0\}) & \text{if } t < 1 \\ \max(wl^\alpha_e(t-1) + Den_e(\{r_{ij} : T_{ij} = t\}) - \alpha, 0) & \text{otherwise} \end{cases} \\
wl^\alpha(t) &= \max_e wl^\alpha_e(t)
\end{aligned}$$

We say that a network with small edge separators is α-overloaded as long as it has no chance to finish its α-workload, i.e. its α-workload is greater than 0.

Definition 7.2 *A network with small edge separators is* α-overloaded *at time t if $wl^\alpha(t) > 0$*

Based on these definitions one can view any schedule in retrospect and define periods of α-overload. Let $T_0^\alpha, \ldots, T_m^\alpha$ be all the disjoint maximal intervals of α-overloaded time steps; T_0^α is the first maximal interval and T_i^α is the i-th one. In addition let us define a function $\tau^\alpha(t)$ which maps a time step t to the first time step $t' \geq t$ at which the network is not overloaded.

As mentioned earlier, the density of a set of requests is a good lower bound on the performance of any optimal schedule on networks with small edge separators. Actually, we argue that our algorithms GREEDY and STEADY are k-competitive by showing that the length of the generated schedules is within a factor of k of the density for any set of requests. Now we bound the maximum response time of the network for the on-line call admission algorithm STEADY. The proof is based on two of STEADY's properties.

Theorem 7.3 *Let $0 < \alpha < 1$ be such that STEADY is an on-line $1/\alpha$-competitive call admission algorithm with respect to makespan in the desired model. Then the maximum response time of any request arriving at time step t is less than $\tau^\alpha(t) - t$, which is less than the length of the current α-overloaded period.*

Proof. Section 5.2 shows a stronger result than just competitiveness: it shows that the length of the schedule produced by STEADY is less than $\textsc{Den}(R)/\alpha$. Thus if follows from the definition of α-workload that at the end of a period of α-overload STEADY will have scheduled all jobs. So the delay of any job presented at time t is less than $\tau^\alpha(t) - t$. □

More generally, we can bound the response time of competitive call admission algorithms if two conditions hold. The algorithm has to be designed using the batch system based on Shmoys, Wein and Williamson [SWW91] (see Section 6) and the k-competitiveness proof has to show that for any set of requests the length of the makespan of the algorithm is at most a factor k worse than the density.

Theorem 7.4 *Given a tree network and a $1/(2\alpha)$-competitive batch-style call admission algorithm A then the fully on-line call admission algorithm designed from A using the batch system, guarantees that the maximum response time of any request arriving at time step t is less than $\tau^\alpha(t) - t$.*

Proof.

Assume that r is the first request arriving at time step t with response time greater than $\tau^\alpha(t) - t$. Then the workload at time step t has to be less than $wl^\alpha(t) \leq (\tau^\alpha(t) - t) * \alpha$. In addition the actual density at time $\tau^\alpha(t)$ of the set of requests R' with arrival times between t and $\tau^\alpha(t)$ has to be less than or equal to α times $\tau^\alpha(t) - t$; $\textsc{Den}(R', \tau^\alpha(t)) \leq (\tau^\alpha(t) - t) * \alpha$. Now since the use of the batch system costs us at most a factor of 2 (see Section 6) and A guarantees that for any set of requests its makespan is only a $1/(2\alpha)$ factor worse than the density the request r cannot have been scheduled at a time greater than $\tau^\alpha(t)$. Therefore its response time is less than than $\tau^\alpha(t) - t$ a contradiction to the assumption. □

For GREEDY this leads to the following Corollary.

Corollary 7.5 *Given a tree network a variant of GREEDY guarantees that the maximum response time of any request arriving at time step t is less than $\tau^{\alpha}(t) - t$ for $\alpha = 1/(4(c+1)\log_2 n)$.*

This is the best one can hope for modulo α, since if all requests of an overloaded period arrived at the beginning of the period even an optimal algorithm cannot guarantee a better response time.

If not all requests arrive at the beginning of the overloaded period, one can do slightly better for batch systems. Again one can look at the generated schedule in retrospect and let l_i be the time the i-th batch finished. The response time for any request arriving, while batch i is scheduled, is bounded by the time it takes to schedule batch i and batch $i+1$. If for any batch i the density of the requests which arrived during this period (between time l_i and l_{i+1}) is at most c times the density of the requests of the batch $i-1$ (the work arrived between time l_{i-1} and l_i) then the maximum response time of any requests is bounded by $c+1$ times the length of the schedule for the current batch. This implies that if a user can monitor the current response times and knows that the workload will not change dramatically he can estimate the response time of his connection request.

The most important conclusion of the section is that GREEDY and STEADY do not introduce any delays unnecessarily. As long as the network is lightly loaded the response time is minimal. Only in the case of overload the algorithms explore the possibility of delaying requests. And even then we can still bound the maximum response time. In addition maximizing the network utilization in the case of overload by minimizing the makespan has the additional benefit of keeping periods of overload as short as possible and therefore helps the network to return to normal operation as quickly as possible. So we should expect that these algorithms behave reasonably in practice.

8 Concluding remarks

In this paper we show that the ability to delay a request by a constant times its duration gives no advantage to a scheduling algorithm with respect to both call-admission ratio and data-admission ratio. It is an open question if there exists an alternative way to limit the delay that actually gives an advantage to the algorithm.

Our non-constant lower bound with respect to makespan for tree networks shows that there is a significant difference between scheduling on trees and scheduling on linear array networks. The gap between our lower bound and the $\Theta(\log n)$ upper bound of GREEDY on trees raises the following questions: are there better algorithms for trees, and can the lower bound be strengthened? Another point of interest is whether the lower bound can be extended to hold against randomized algorithms.

We prove GREEDY on the linear array to be between 4.4 and 25.8-competitive with respect to makespan for arbitrary durations. Although we present algorithm STEADY to narrow this gap we still don't know GREEDY's real competitive ratio nor whether it holds for arbitrary bandwidth. And no algorithm has been found with a provably optimal competitive ratio.

Finally, it would be interesting to extend our framework for analyzing maximum response time based on the context of network overloading to a more general environment.

References

[AAP93] Baruch Awerbuch, Yossi Azar, and Serge Plotkin. Throughput-competitive on-line routing. In *Proceedings of the 34th Annual Symposium on Foundations of Computer Science*, pages 32–41, Palo Alto, California, Nov 1993.

[AAPW94] Baruch Awerbuch, Yossi Azar, Serge Plotkin, and Orli Waarts. Competitive routing of virtual circuits with unknown durations. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 321–327, Arlington, Virginia, Jan 1994.

[ABFR94] Baruch Awerbuch, Yair Bartal, Amos Fiat, and Adi Rósen. Competitive non-preemptive call control. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 312–320, Arlington, Virginia, Jan 1994.

[AGLR94] Baruch Awerbuch, Rainer Gawlick, Tom Leighton, and Yuval Rabani. On-line admission control and circuit routing for high performance computing and communication. In *Proceedings of the 35st Annual Symposium on Foundations of Computer Science*, pages 412–423, Santa Fe, New Mexico, November 1994.

[AW92] Heinrich Armbrüster and Klaus Wimmer. Broadband multimedia applications using ATM networks: High-performance computing, high-capacity storage, and high-speed communication. *IEEE Journal on selected areas in communications*, 10(9):1382–1396, Dec 1992.

[CS88] M. Chrobak and Maciej Slusarek. On some packing problems related to dynamic storage allocation. *RAIRO Informatique Thèorique et Applications*, 22(4):487–499, 1988.

[GGK+93] J.A. Garay, I. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient on-line call control algorithms. In *Proceedings of 2nd Annual Israel Conference on Theory of Computing and Systems*, pages 285–293, Los Alamitos, California, June 1993.

[Kie88] H.A. Kierstead. The linearity of first-fit for coloring interval graphs. *SIAM Journal of Discrete Mathematics*, 1(4):526–530, 1988.

[KQ92] H.A. Kierstead and Jun Qin. First-fit and interval graphs, 1992.

[LT79] Richard J. Lipton and Robert Endre Tarjan. A separator theorem for planar graphs. *SIAM Journal on Applied Mathematics*, 36(2):177–189, 1979.

[LT94] Richard J. Lipton and Andrew Tomkins. Online interval scheduling. In *Proceedings of the 5th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 302–311, Arlington, Virginia, Jan 1994.

[Slu89] Maciej Slusarek. A coloring algorithm for interval graphs. In *Mathematical Foundations of Computer Science 1989 Proceedings*, pages 471–480, Berlin, West-Germany, 1989.

[ST85] Daniel D. Sleator and Robert E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2):202–208, 1985.

[SWW91] David B. Shmoys, Joel Wein, and David P. Williamson. Scheduling parallel machines on-line. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science*, pages 131–140, San Juan, Puerto Rico, October 1991.