On-Line Routing of Virtual Circuits with Applications to Load Balancing and Machine Scheduling*

James Aspnes[†] *Yossi Azar*[‡] *Amos Fiat*[§]

Serge Plotkin[¶] *Orli Waarts*[||]

Abstract

In this paper we study the problem of on-line allocation of routes to virtual circuits (both *point-to-point* and *multicast*) where the goal is to route all requests while minimizing the required bandwidth. We concentrate on the case of *permanent* virtual circuits (*i.e.,* once a circuit is established, it exists forever), and describe an algorithm that achieves an $O(\log n)$ competitive ratio with respect to maximum congestion, where n is the number of nodes in the network. Informally, our results show that instead of knowing all of the future requests, it is sufficient to increase the bandwidth of the communication links by an $O(\log n)$ factor. We also show that this result is tight, *i.e.* for any on-line algorithm there exists a scenario in which $\Omega(\log n)$ increase in bandwidth is necessary in directed networks.

We view virtual circuit routing as a generalization of an on-line load balancing problem, defined as follows: jobs arrive on line and each job must be assigned to one of the machines immediately upon arrival. Assigning a job to a machine increases this machine's load by an amount that depends both on the job and on the machine. The goal is to minimize the maximum load.

For the *related machines* case, we describe the first algorithm that achieves constant competitive ratio. For the *unrelated* case (with n machines), we describe a new method that yields $O(\log n)$-competitive algorithm. This stands in contrast to the natural greedy approach, whose competitive ratio is exactly n.

*A preliminary version of this paper was presented at the 1993 ACM Symposium on Theory of Computing.

†Department of Computer Science, Yale University. During part of this research this author was visiting IBM Almaden Research Center.

‡Department of Computer Science, Tel-Aviv University, Israel. Portion of this work was done while the author was at Digital - Systems Research Center. Supported by an Alon fellowship and a grant from the Israel Science Foundation, administered by the Israel Academy of Sciences. E-Mail: azar@math.tau.ac.il

§Department of Computer Science, Tel-Aviv University, Israel. Supported by a grant from the Israel Science Foundation, administered by the Israel Academy of Sciences. E-mail: fiat@math.tau.ac.il

¶Dept. of Computer Science, Stanford University. Research supported by ARO Grant DAAH04-95-1-0121, NSF Grant CCR-9304971, and Terman Fellowship. E-Mail: plotkin@cs.stanford.edu.

||IBM Almaden Research Center. Part of this research was done while the author was with Stanford University, supported by U.S. Army Res. Office Grant DAAL-03-91-G-0102. E-Mail: orli@cs.stanford.edu.

1 Introduction

Virtual Circuit Routing High-speed integrated communication networks are going to become a reality in the near future. Implementation of these networks raises numerous new issues that either did not exist or could be easily addressed in the context of the existing low-speed networks. In particular, the increase in the network speed by several orders of magnitude leads to a situation where a small delay of a high-rate bit stream quickly exceeds the available buffer space. This makes it advantageous to use bandwidth-reservation techniques.

The main abstraction through which the customer can use the network is by a ***virtual circuit***. In order to use the network, the customer requests it to reserve the required bandwidth between the two communicating points. The network guarantees that the reserved bandwidth will indeed be available as long as needed, creating an illusion of a real circuit dedicated to the customer. One of the basic services that appears in the proposals for future high-speed networks (***e.g.*** ATM [1]) is the ***permanent virtual circuit*** (PVC) service. As far as the user is concerned, such virtual circuit is supposed to behave like a physical line connecting the corresponding points, and hence it is desirable that once such a circuit is created, it will not be "rerouted" by the network except as a result of failures. (Hence the name "permanent".)

In this paper we develop a framework and techniques that allow us to address the problem of ***online virtual circuit routing***. We consider the following idealized setting: We are given a network where each edge has an associated capacity (bandwidth). Requests for virtual circuits arrive on line, where each request specifies the source and destination points, and the required bandwidth. The routing algorithm has to choose a path from the source to the destination and reserve the required bandwidth along this path. The goal is to minimize the maximum (over all edges) of the relative load, defined as the reserved (used) edge bandwidth, measured as a percentage of the total edge capacity. In this paper, we assume that no rerouting is allowed and that the virtual circuits never disappear. Because all requests must be routed, it follows that the relative load may exceed 1, which can be viewed as a slowdown.

For some applications it is more efficient to use ***multicast*** circuits, where instead of a single destination there are multiple destinations. Examples include teleconferencing, video on demand, database updates, etc. In this case the routing algorithm has to choose a tree that spans the nodes participating in the multicast.

Our framework and techniques can be applied to several alternative models; discussion of these models is deferred to the end of the introduction. Several recent papers show how to extend the techniques developed in this paper to more general cases, including the routing of ***switched virtual circuits*** (SVC), ***i.e.*** circuits that have limited duration in time [11, 6], concurrent routing [2], routing with limited rerouting [7], and min-cost circuit routing [4]. Multicast circuits were analyzed in [3]. Analysis of routing in a Poisson arrivals model was done in [21]. Simulation and implementation results described in [17, 18] indicate that online routing algorithms based on our techniques outperform traditional algorithms for routing virtual circuits in ATM networks.

As customary, we evaluate the performance of the on-line algorithms in terms of ***competitive ratio***, introduced in [36] and further developed in [24, 14, 30]. In our case, it corresponds to the supremum, over all possible input sequences, of the ratio of the maximum relative load achieved by the on-line algorithm to the maximum relative load achieved by the optimal off-line algorithm.

Using our framework, we derive online virtual circuit routing algorithms (point-to-point and multicast) that are $O(\log n)$ competitive with respect to load, where n is the number of nodes in the network. We also show an $\Omega(\log n)$ lower bound on the competitive ratio of any virtual circuit routing algorithm in the case where the underlying network is directed. (The upper bound works for both directed and undirected cases.)

Load Balancing We view virtual circuit routing as a generalization of on-line machine scheduling/load balancing. To this end, we concentrate on ***non-preemptive*** load-balancing, defined as follows: There are n parallel machines and a number of independent jobs; the jobs arrive one by one, where each job has an associated ***load vector*** and has to be assigned to exactly one of the machines, thereby increasing the ***load*** on this machine by the amount specified by the corresponding coordinate of the load vector. Once a job is assigned, it cannot be re-assigned. The objective is to minimize the maximum load.

The load balancing problems can be categorized into three classes according to the properties of the load vectors, as it is done for the non-preemptive scheduling problems [20]. In the ***identical*** machines case, all the coordinates of a load vector are the same. This case was first considered by Graham [19], who showed a $(2-\frac{1}{n})$-competitive algorithm, where n is the number of machines. The bound was improved in [13] to $2-\epsilon$ for a small constant ϵ (the value of ϵ was further improved in [22]). In the ***related*** machines case, the ith coordinate of each load vector is equal to $w(j)/v(i)$, where the "weight" $w(j)$ depends only on the job j and the "speed" $v(i)$ depends only on the machine i. All other cases are referred to as ***unrelated machines***. The special case where all coordinates of the load vector are either ∞ or equal to a given value that depends only on the job, was considered in [12], who described an $O(\log n)$-competitive algorithm. This case can be viewed as a hybrid between the identical and the unrelated machines case, and it is incomparable to the related machines case. A similar special case was studied in [25].

In this paper we show an $O(\log n)$-competitive algorithm for the unrelated machines case, and an 8-competitive algorithm for the related machines case. Although competitive analysis notions apply to algorithms without any restrictions on their running times, all on-line algorithms presented in this paper run in deterministic polynomial time, whereas the matching lower bounds are based on information-theoretic arguments and apply even if we allow the online algorithm to use randomization.

The ***related machines*** case is a generalization of the identical machines problem, for which Graham [19] has shown that a greedy algorithm achieves a constant competitive ratio. Thus, it is natural to ask whether an adaptation of such an algorithm can give a constant competitive ratio for the related machines case as well. We prove that the natural greedy approach (that is, assigning every new job to the machine that will complete it with the lowest resulting load) is $\Theta(\log n)$ competitive. Our (non-greedy) 8-competitive algorithm for this problem can be viewed as an adaptation of the scheduling algorithm of Shmoys, Wein, and Williamson [35] to the context of load balancing.

We show that for the ***unrelated machines*** case, the natural greedy algorithm is exactly n-competitive. This bound should be contrasted with the optimal $O(\log n)$-competitive greedy strategy of [12] for the special case where all coordinates of the load vector are either ∞ or equal to a given value that depends only on the job. Thus, the unrelated machines case requires development of new techniques. We introduce a new approach that leads to an $O(\log n)$-competitive algorithm for the general unrelated machines case. As shown in [12], this is the best bound on the competitive ratio one can hope for in this case.

It is easy to see that the identical and related machines problems are special cases of virtual circuit routing. These problems can be reduced to a virtual circuit routing problem on a 2 vertex network, with multiple edges between them. Every edge represents a machine, and the edge capacities represent the relative machine speeds. Every arriving job is translated into a call between the two vertices s and t where the call bandwidth corresponds to the job weight. The unrelated machine is a special case of a *generalization* of virtual circuit routing where the call requires different bandwidth depending on the edge used. Our virtual circuit algorithms give an optimal $O(\log n)$ competitive ratio for this problem.

Other related work Some of the techniques which are used for our on-line framework are based on ideas developed in the context of approximation algorithms for the multicommodity flow and related problems (see ***eg.*** [34, 26, 28, 33, 23]). In particular, we assign each link a weight that is exponential in the link's load, and choose the routes by computing shortest paths with respect to this weight. The main difference between the algorithms presented here and the previously known offline approximation algorithms is in a novel way of proving the approximation factor which allows us to execute the algorithm in an online fashion.

All the results in this paper concentrate on the case where jobs and virtual circuits are permanent, ***i.e.*** jobs never leave and virtual circuits never terminate. Azar, Broder, and Karlin [10] introduced a natural generalization of this model, in which requests have ***duration*** in time. They show an $\Omega(\sqrt{n})$ lower bound on the competitive ratio of any load balancing algorithm that deals with the ***unknown duration*** case, ***i.e.*** the case where the duration of a request becomes known only upon its termination.

This lower bound suggests considering the case where the duration of a request becomes known upon its arrival (the ***known duration*** case). The methods developed in this paper were generalized in [11], giving an $O(\log nT)$-competitive algorithm for the problems of scheduling unrelated machines in the known duration case, where T is the ratio of maximum to minimum duration. Similar results can be achieved for the virtual circuit routing problem. Recently, the lower bound in [10] was simplified and improved by Ma and Plotkin [29]. In particular, their bound implies that we can not expect to have a $poly(lognT)$ competitive algorithm for the unknown duration case.

Another way to overcome the lower bound in the unknown-duration case is to allow reassignments of existing jobs. For the case where the coordinates of the load vector are restricted to be 1 or ∞, Phillips and Westbrook [31] proposed an algorithm that achieves $O(\log n)$ competitive ratio while making $O(1)$ amortized reassignments per job. The general case was considered in [7], who show how to extend the techniques presented here to design an $O(\log n)$-competitive algorithm with respect to load that reroutes each circuit at most $O(\log n)$ times.

An alternative measure of network performance is the amortized throughput defined as the average over time of the number of bits transmitted by the accepted connections. In this setting, the network's bandwidth is assumed to be insufficient to satisfy all the requests so some of the requests may need to be rejected upon their arrival. An on-line algorithm in this setting is a combination of a decision mechanism that determines which requests to satisfy together with a strategy that specifies how to route these requests. The goal is to maximize the amortized throughput. A competitive algorithm that maximizes the throughput in a single-link case was provided by Garay and Gopal [16]; the case where the network consists of single line of nodes was considered by Garay, Gopal, Kutten, Mansour and Yung in [15]. The techniques presented here were extended by Awerbuch, Azar, and Plotkin [6] to provide competitive solutions for networks with unrestricted topology. Other studies on the throughput performance measure appear in [9, 8, 27]. Our routing and scheduling algorithms assume a central scheduler that makes all the decisions. In [2], Awerbuch and Azar extended the techniques of this paper to the case where there are concurrent requests that have to be satisfied in a decentralized fashion. See [32] for a survey of different online routing strategies. We note that another measure of performance for load balancing algorithms appears in [5] who provided competitive algorithms for that measure.

2 Virtual Circuit Routing

In this section we consider the problem of on-line routing of virtual circuits in a capacitated network. Formally, we are given a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, and a capacity function $u : E \to \mathbb{R}^+$. The requests arrive as tuples $(s_i, t_i, p(i))$, where $s_i, t_i \in V$ and $p(i) \in \mathbb{R}^+$. Request i is satisfied by choosing a route P_i from s_i to t_i and reserving capacity $p(i)$ along this route.

Since we will always normalize the requested bandwidth to the total available bandwidth, it will be convenient to define:

$$\forall e \in E, p_e(i) = p(i)/u(e).$$

Let $\mathcal{P} = \{\mathcal{P}_\infty, \mathcal{P}_\in, \ldots, \mathcal{P}_{||}\}$ be the routes assigned to requests 1 through k by the on-line algorithm, and let $\mathcal{P}^* = \{\mathcal{P}^*_\infty, \mathcal{P}^*_\in, \ldots, \mathcal{P}^*_{||}\}$ be the routes assigned by the off-line algorithm. Given a set of routes $\mathcal{P}$, define the ***relative load*** after the first j requests are satisfied by

$$\ell_e(j) = \sum_{\substack{i: e \in P_i \\ i \leq j}} p_e(i) \tag{1}$$

and let $\lambda(j) = \max_{e \in E} \ell_e(j)$. Similarly, define $\ell^*_e(j)$ and $\lambda^*(j)$ to be the corresponding quantities for the routes produced by the off-line algorithm. For simplicity we will abbreviate $\lambda(k)$ as λ

```
procedure ASSIGN-ROUTE(p, s, t, G, ℓ⃗, Λ, β);
    /* Λ — current estimate of L*.
    /* β — designed performance guarantee of the algorithm.
    ∀e ∈ E, p_e := p/u(e);
    ∀e ∈ E, p̃_e := p_e/Λ;
    ∀e ∈ E, ℓ̃_e := ℓ_e/Λ;
    ∀e ∈ E : c_e := a^(ℓ̃_e + p̃_e) − a^(ℓ̃_e);
    Let P be the shortest path from s to t in G w.r.t. costs c_e;
    if ∃e ∈ P : ℓ_e + p_e > βΛ
      then b :=fail
      else begin
        ∀e ∈ P : ℓ_e := ℓ_e + p_e;
        b :=success
      end;
    return(P, ℓ⃗, b).
end.
```

Figure 1: Algorithm ASSIGN-ROUTE.

and $\lambda^*(k)$ as λ^*. The goal of the on-line algorithm is to produce a set of routes $\mathcal{P}$ that minimizes λ/λ^*.

This problem can be viewed as an instance of 2-terminal net routing or ***path-packing***. Minimizing λ/λ^* corresponds to asking how much larger we should make the capacities of the edges in order for the on-line algorithm to be able to satisfy all the requests that the off-line algorithm could have satisfied in the network with the original capacities.

It is easy to see that the algorithms presented in this section can be extended to the case where the increase in the bandwidth is not uniform along the route. Namely, the case where a routing request is a vector giving the capacity requirement of the call on each and every edge in the network, should that edge we used in the route. The goal here is to present a connected path of edges from source to destination, where the capacity used on the edges for this call is obtained from the input vector. This is clearly a generalization of the standard routing problem where the entry associated with all edges e is simply the (single) call bandwidth.

2.1 Routing Algorithm

The ASSIGN-ROUTE algorithm is shown in Figure 1. Given a request to allocate a route of capacity p from s to t, ASSIGN-ROUTE assigns a weight to each edge as a function of the change in its relative load that would occur if it were to be used by the new route, and then computes a shortest path from s to t with respect to these weights; a is an appropriately chosen constant.

For convenience, we define the notion of a ***designed performance guarantee*** β as follows: the algorithm accepts a parameter Λ and never creates load that exceeds $\beta\Lambda$. The algorithm is allowed to return "fail" and to refuse to route a circuit if $\Lambda < \lambda^*$, otherwise it has to route all of the requests.

Lemma 2.1 **If $\lambda^* \leq \Lambda$, then there exists $\beta = O(\log n)$ such that algorithm Assign-Route never fails. Thus, the relative load on an edge never exceeds $\beta \cdot \Lambda$.**

Proof: To simplify the formulas, we will use tilde to denote normalization by Λ, for example $\tilde{\ell}_e(j) = \ell_e(j)/\Lambda$. Define the potential function:

$$\Phi(j) = \sum_{e \in E} a^{\tilde{\ell}_e(j)}(\gamma - \tilde{\ell}^*_e(j)), \tag{2}$$

where $a, \gamma > 1$ are constants. Note that Φ is a function of both $\tilde{\ell}_e$ and $\tilde{\ell}_e{}^*$. If the on-line algorithm satisfies the $(j+1)$st request with route P_{j+1} and the off-line algorithm satisfies it with route P^*_{j+1}, we get the following change in the potential function:

$$\begin{aligned}
\Phi(j+1) - \Phi(j) &= \sum_{e \in P_{j+1}} (\gamma - \tilde{\ell}^*_e(j))(a^{\tilde{\ell}_e(j+1)} - a^{\tilde{\ell}_e(j)}) - \sum_{e \in P^*_{j+1}} a^{\tilde{\ell}_e(j+1)} \tilde{p}_e(j+1) \\
&\leq \sum_{e \in P_{j+1}} \gamma(a^{\tilde{\ell}_e(j)+\tilde{p}_e(j+1)} - a^{\tilde{\ell}_e(j)}) - \sum_{e \in P^*_{j+1}} a^{\tilde{\ell}_e(j)} \tilde{p}_e(j+1) \\
&\leq \sum_{e \in P^*_{j+1}} \left(\gamma(a^{\tilde{\ell}_e(j)+\tilde{p}_e(j+1)} - a^{\tilde{\ell}_e(j)}) - a^{\tilde{\ell}_e(j)} \tilde{p}_e(j+1)\right) \\
&= \sum_{e \in P^*_{j+1}} a^{\tilde{\ell}_e(j)} \left(\gamma(a^{\tilde{p}_e(j+1)} - 1) - \tilde{p}_e(j+1)\right)
\end{aligned} \tag{3}$$

The first equality can be viewed as an application of the identity $(x + \Delta(x))(y + \Delta(y)) = y\Delta(x) + (x + \Delta(x))\Delta(y)$, where $x = a^{\tilde{\ell}_e(j)}$, $y = (\gamma - \tilde{\ell}^*_e(j))$, $\Delta(x)$ and $\Delta(y)$ represent the changes in the value of these terms between $\Phi(j+1)$ and $\Phi(j)$. The first inequality follows because $\tilde{\ell}^*_e(j)$ is non-negative for all e. The last inequality follows from the fact that P_{j+1} is the shortest path between the endpoints of the $j+1$st request with respect to the costs $a^{\tilde{\ell}_e(j)+\tilde{p}_e(j+1)} - a^{\tilde{\ell}_e(j)}$.

Since the $(j+1)$st request is satisfied by the optimal algorithm by assigning it the route P^*_{j+1}, it means that $\forall e \in P^*_{j+1} : \; 0 \leq \tilde{p}_e(j+1) \leq \lambda^*/\Lambda \leq 1$. Therefore, in order to show that the potential function does not increase, it is sufficient to show that $\forall x \in [0,1] : \quad \gamma(a^x - 1) \leq x$, which is true for $a = 1 + 1/\gamma$.

Initially, $\Phi(0) \leq \gamma m$, where m is the number of edges in the graph. Since Φ does not increase, and since $\tilde{\ell}^*(j) \leq 1$, then after satisfying k requests, we have:

$$\sum_e (\gamma - 1) a^{\frac{\ell_e(k)}{\Lambda}} \leq \gamma m.$$

The last inequality, and the fact that $\gamma > 1$ imply

$$\max_{e \in E} \ell_e(k) \leq \Lambda \log_a \left(\frac{\gamma m}{\gamma - 1} \right) = O(\Lambda \log n).$$

▮

We use a simple doubling technique to overcome the problem that Λ in unknown. This causes the competitive ratio to be a factor of 4 larger than the designed performance ratio.

The algorithm works in phases, where the difference between phases is the value of Λ assumed by the algorithm. Within a phase the algorithm Assign-Route is used to route calls, ignoring all calls routed in previous phases. The first phase has $\Lambda = \min_e p_e(1) = \min_e p(1)/u(e)$, where $p(1)$ is the bandwidth request of the first call. At the beginning of every subsequent phase the value of Λ doubles. A new phase starts when Assign-Route returns "fail". As mentioned above, Assign-Route will ignore all calls routed in previous phases.

It is easy to see that this approach can increase the competitive factor by at most a factor of 4 (a factor of 2 due to the load in all the rest of the phases except the last, and another factor of 2 due to imprecise approximation of λ^* by Λ). Since the designed performance guarantee of Assign-Route is $O(\log n)$, we get the following theorem (observe that it holds both for directed and undirected graphs):

Theorem 2.2 Algorithm Assign-Route can be used to achieve $O(\log n)$ competitive ratio with respect to load.

2.2 Routing Multicast Circuits

Many applications (teleconferencing, video on demand, etc.) are based on ***multicast*** instead of ***point-to-point*** circuits. A request for a multicast circuit consists of a tuple $(t_i^1, t_i^2, \ldots t_i^{k_i}, p(i))$ where $t_i^1, t_i^2, \ldots, t_i^{k_i}$ are the communicating points and $p(i)$ is the required bandwidth. Any one of the communicating points can serve as a "source". To satisfy such request, the algorithm needs to assign the required bandwidth $p(i)$ along the edges of some tree T_i that spans nodes $t_i^1, t_i^2, \ldots, t_i^{k_i}$. As in the point-to-point case considered in the previous section, the goal is to minimize load. Also note that the case $k_i = 2$ directly corresponds to the point-to-point case.

The algorithm to route multicast circuits is a direct extension of the point-to-point routing strategy presented above. Instead of routing over ***min-weight paths***, multicast circuits are routed over ***min-weight Steiner trees***.

As finding min-weight Steiner tress is NP-hard, we actually approximate them. However, imagine for now that we were to route the multicast calls along the Steiner trees. The proof of the competitive ratio is nearly identical to the proof of Lemma 2.1. The only difference is in the summation range in Equation (3). Instead of summing over the edges on the path chosen by the algorithm and the edges on the optimum path, the sum will be over the edges of the ***tree*** chosen by the algorithm and the edges of the tree chosen by the optimum offline algorithm.

Equation (3) is based on the fact that the cost of the edges chosen by the algorithm for routing of the current circuit is not larger than the cost of edges chosen by the optimum algorithm for routing of this circuit. The fact that these edges form a path is not used. In other words, Equation (3) remains correct if the summation range is changed and if the algorithm routes over min-weight Steiner trees. Thus, a multicast algorithm that routes over min-weight Steiner trees is $O(\log n)$-competitive.

Instead of routing the multicast call over the min-weight Steiner tree, we route it over a tree that is a constant factor approximation to the min-weight Steiner tree. Such trees can be found by applying a min-cost spanning tree algorithm over an appropriately constructed graph [37].

If we use a 2-approximation to the min-weight steiner tree then in the 3rd and 4th lines of Equation (3) the value γ is replaced by 2γ. However, we can ensure that $2\gamma(a^x - 1) \leq x$ is true by choosing $a = 1 + 1/(2\gamma)$ (replacing the previously used $a = 1 + 1/\gamma$).

The above discussion implies the following claim:

Theorem 2.3 There exists an $O(\log n)$-competitive algorithm for multicast virtual circuit routing, where each decision can be implemented in polynomial time.

2.3 Lower Bound for Routing

In this section we show a lower bound of $\Omega(\log n)$ for the competitive ratio of any on-line routing algorithm in a directed network, *i.e.* a network where the capacity between v and w is not necessarily equal to the capacity between w and v. This implies that our $O(\log n)$-competitive algorithm presented in the previous section is optimal in this case. Our lower bound also holds for randomized algorithms working against an oblivious adversary, *i.e.* an adversary that has to generate new requests independently of the outcome of the coin flips of the online algorithm. The basic idea is to modify the lower bound of Azar, Naor and Rom [12] for on-line load balancing.

Without loss of generality, assume that n is a power of 2. Consider a directed graph that has a single source s, connected to each one of n vertices $v_1, v_2, \ldots, v_n$. There is one sink, denoted by $S_{1,1}$, connected to $v_1, \ldots, v_n$; two sinks, denoted by $S_{2,1}, S_{2,2}$, connected to $v_1, \ldots, v_{n/2}$ and $v_{n/2+1}, \ldots, v_n$, respectively, etc. In general, for each $1 \leq i \leq \log n$, we divide vertices $v_1, \ldots, v_n$ into 2^{i-1} sets, the jth of which, for $j = 1, \ldots, 2^{i-1}$, contains vertices $v_{(j-1)n/2^{i-1}+1}, \ldots, v_{jn/2^{i-1}}$. Each of the vertices in a set is connected to a sink associated with this set, where the sinks are denoted by $S_{i,j}$ for $j = 1, \ldots, 2^{i-1}$. Observe that the vertices associated with $S_{i,j}$ are the union of two disjoint sets associated with $S_{i+1,2j-1}$ and $S_{i+1,2j}$.

We construct a sequence of requests for paths from the source to the sinks, for which the off-line load is at most 1 but the online algorithm assigns at least load $\frac{\log n}{2}$ to some edge (s, v_j). This, combined with the fact that the size of the graph is $O(n)$, will yield the $\Omega(\log n)$ lower bound.

We will refer to the load on an edge (s, v_j) as the ***load of*** v_j. Requests are generated in $\log n$ phases; the bandwidth of every request is equal to 1. We will maintain that the following conditions hold for each phase $1 \leq i \leq \log n$:

1. In phase i, there are $\frac{n}{2^i}$ requests of paths from the source to a sink $S_{i,j}$ for some j, where $1 \leq j \leq 2^{i-1}$.
2. At the end of phase i, the average of the expected load, over the vertices v_ℓ, associated with sink $S_{i,j}$ is at least $i/2$.

Clearly, before the first phase begins, the load of each vertex is 0. Assume that the above conditions hold for phase i. The vertices associated with $S_{i,j}$ are the union of two disjoint sets: vertices associated with sink $S_{i+1,2j-1}$, and those associated with $S_{i+1,2j}$. Hence, one of these sets must have average expected load of $i/2$ at the end of the ith phase. Denote this subset by S. Generate $\frac{n}{2^{i+1}}$ requests for unit capacity from source s to the sink associated with S. Since S is of size $\frac{n}{2^i}$, the average expected load of S must increase by $1/2$, to at least $(i+1)/2$, implying that the conditions are satisfied for phase $i+1$.

Thus, after the last phase, the average expected load of the two vertices in the last set is at least $\frac{\log n}{2}$. Hence the expected load of one of them is at least $\frac{\log n}{2}$.

To complete the proof, we have to show that the off-line algorithm can maintain unit maximum load. It is enough to show that at each phase the off-line can satisfy the requests by using edge-disjoint paths and without using vertices associated with sinks requested in latter phases. Indeed, at phase i there are $\frac{n}{2^i}$ requests for paths from the source to some sink $S_{i,j}$. The set of vertices associated with this sink contains two disjoint sets each of size $\frac{n}{2^i}$. By construction, one of these sets is not associated with sinks of latter requests. Thus, the off-line algorithm can route all the requests of phase i through edge-disjoint paths that use only vertices of that set.

3 Online Machine Load-Balancing

In this section we present several algorithms for online machine load-balancing. Jobs arrive online, and each has to be immediately assigned to one of the machines. The goal is to minimize maximum load.

Formally, each job j is represented by its "load vector" $\vec{p}(j) = (p_1(j), p_2(j), \ldots, p_n(j))$, where $p_i(j) \geq 0$. Assigning job j to machine i increases the load on this machine by $p_i(j)$. Let $\ell_i(j)$ denote the load on machine i after we have already assigned jobs 1 through j:

$$\ell_k(j) = \begin{cases} \ell_k(j-1) + p_k(j) & \text{if } k = i \\ \ell_k(j-1) & \text{otherwise} \end{cases}$$

Consider a sequence of jobs defined by $\sigma = (\vec{p}(1), \vec{p}(2), \ldots, \vec{p}(k))$. Denote by $\ell_i^*(j)$ the load on machine i achieved by the off-line algorithm $\mathcal{A}^*$ after assigning jobs 1 through j in σ. The goal of both the off-line and the on-line algorithms is to minimize $L^*(k) = \max_i \ell_i^*(k)$ and $L(k) = \max_i \ell_i(k)$, respectively. More precisely, we measure the performance of the on-line algorithm by the supremum over all possible sequences of $L(k)/L^*(k)$ (of arbitrary length k).

As we have mentioned in the Introduction, the load-balancing problems are usually categorized into three classes, based on the properties of the load vectors. For ***identical*** machines, $\forall i, i', j: \ p_i(j) = p_{i'}(j)$. For ***related*** machines, $\forall i, i', j, j': \ p_i(j)/p_{i'}(j) = p_i(j')/p_{i'}(j') = v_{i'}/v_i$, where v_i denotes the speed of machine i. All other cases are referred to as ***unrelated*** machines.

Note that instead of "load" one can talk about "execution time". Restating the problem in these terms, our goal is to decrease maximum execution time under the requirement that the arriving jobs are ***scheduled immediately.***

3.1 Unrelated Machines

In this section we consider on-line load-balancing on ***unrelated machines***. As we will show in Section 4, the natural greedy approach is far from optimal for this case, achieving a competitive ratio of $\Theta(n)$.

An $O(\log n)$-competitive algorithm for the unrelated machines load-balancing can be constructed as a reduction to the routing problem considered in the previous section. Unfortunately, such reduction results in a confusing and non-intuitive algorithm. Instead, we present a simpler algorithm, specifically designed for the machine load-balancing problem.

For simplicity, we first consider the case where we are given a parameter Λ, such that $\Lambda \geq L^*$. As before, an appropriate value of Λ can be "guessed" using a simple doubling approach, increasing the competitive ratio by at most a factor of 4. We use tilde to denote normalization by Λ, ***i.e.*** $\tilde{x} = x/\Lambda$.

Algorithm Assign-U is shown in Figure 2. The basic step is to assign job j to make $\sum_{i=1}^{n} a^{\tilde{\ell}_i(j)}$ as small as possible. In the description of the algorithm, we have omitted the job index j, since a single invocation of the algorithm deals only with a single job. We will use the notion of ***designed performance guarantee*** similarly to its use in the online routing case: the algorithm accepts a parameter Λ and never creates load that exceeds $\beta\Lambda$. The algorithm is allowed to return "fail" and to refuse to schedule a job if $\Lambda < L^*$, otherwise it has to schedule all of the arriving jobs.

Lemma 3.1 If $\lambda^* \leq \Lambda$, then there exists $\beta = O(\log n)$ such that algorithm Assign-U never fails. Thus, the load on a machine never exceeds $\beta \cdot \Lambda$.

Proof: Consider the state of the system after scheduling $j-1$ jobs, and let $a, \gamma > 1$ be constants. (Later we show that a good choice is $a \approx 2, \gamma \approx 1$.) Recall the assumption that $L^*(j) \leq L^* \leq \Lambda$, and define the potential function:

procedure ASSIGN-U$(\vec{p}, \vec{\ell}, \Lambda, \beta)$;
/* Λ — current estimate of L^*.
/* β — designed performance guarantee of the algorithm.
Let s be the index minimizing $\Delta_i = a^{(\tilde{\ell}_i + \tilde{p}_i)} - a^{\tilde{\ell}_i}$;
if $\ell_s + p_s > \beta\Lambda$
then b :=fail
else begin
$\ell_s := \ell_s + p_s$;
b :=success
end;
return$(\vec{\ell}, b)$.
end.

Figure 2: Algorithm ASSIGN-U.

$$\Phi(j) = \sum_{i=1}^{n} a^{\tilde{\ell}_i(j)}(\gamma - \tilde{\ell}_i^*(j)). \tag{4}$$

Assume that job j was assigned to machine i' by the on-line algorithm and to machine i by the off-line algorithm. Analogously to the proof of Equation (3) we now have:

$$\begin{aligned} \Phi(j) - \Phi(j-1) &= (\gamma - \tilde{\ell}_{i'}^*(j-1))(a^{\tilde{\ell}_{i'}(j)} - a^{\tilde{\ell}_{i'}(j-1)}) - a^{\tilde{\ell}_i(j)}\tilde{p}_i(j) \\ &\leq \gamma(a^{\tilde{\ell}_{i'}(j-1)+\tilde{p}_{i'}(j)} - a^{\tilde{\ell}_{i'}(j-1)}) - a^{\tilde{\ell}_i(j-1)}\tilde{p}_i(j) \\ &\leq \gamma(a^{\tilde{\ell}_i(j-1)+\tilde{p}_i(j)} - a^{\tilde{\ell}_i(j-1)}) - a^{\tilde{\ell}_i(j-1)}\tilde{p}_i(j) \\ &= a^{\tilde{\ell}_i(j-1)}(\gamma(a^{\tilde{p}_i(j)} - 1) - \tilde{p}_i(j)). \end{aligned} \tag{5}$$

Note that since the off-line algorithm has assigned job j to machine i, we have $0 \leq \tilde{p}_i(j) \leq L^*/\Lambda \leq 1$. Therefore, in order to show that the potential function does not increase, it is sufficient to show that $\forall x \in [0,1]: \ \gamma(a^x - 1) \leq x$. This is true for $a = 1 + 1/\gamma$.

Since initially $\Phi(0) = \gamma n$, at any point in the assignment process

$$\sum_{i=1}^{n} a^{\tilde{\ell}_i(j)}(\gamma - 1) \leq \gamma n,$$

and hence

$$\begin{aligned} L &= \max_i \ell_i(k) = \Lambda \max \tilde{\ell}_i(k) \\ &\leq \Lambda \cdot \log_a \left(\frac{\gamma}{\gamma - 1} n \right) = O(\Lambda \log n) \end{aligned} \tag{6}$$

∎

Notice that the constants in the big O of (6) are small; for example for $\gamma = 1.1$, we get $L/\Lambda \leq 1.07 \log n + 3.7$. By changing the value of γ, one can trade off the multiplicative factor against the additive term.

Observe that $\log n$ is a lower bound even for the restricted case considered in [12]. Moreover, it is interesting to note that for the case where the coordinates of the load vector $p(j)$ are either ∞ or equal to some constant p_j that depends only on the job j, our algorithm behaves exactly like the greedy algorithm considered in [12].

3.2 Related Machines

The related machines case is a generalization of the identical machines case. In Section 4 we show that a natural generalization for the related machines case of Graham's greedy algorithm for the identical machines case, leads to an $\Theta(\log n)$ competitive ratio. Here we present a non-greedy algorithm that achieves a constant competitive ratio.

As before, we first consider the case where we are given a parameter Λ, such that $\Lambda \geq L^*(k)$, where k is the index of the last job. A simple doubling technique can be used to eliminate this assumption. Roughly speaking, the algorithm will assign jobs to the slowest machine possible while making sure that the maximum load will not exceed an appropriately chosen bound. The idea of assigning each job to the "least capable machine" first appeared in the paper by Shmoys, Wein, and Williamson [35], where they considered an online scheduling problem.

Algorithm Assign-R is shown in Figure 3. The basic step is to assign job j to the slowest machine such that the load on this machine will be below 2Λ after the assignment. In the description of the algorithm, we have omitted the job index j, since a single invocation of the algorithm deals only with a single job. We assume that the machines are indexed according to increasing speed.

In the following discussion we will omit the index k when it can be understood from the context. In particular, we use L and L^* instead of $L(k)$ and $L^*(k)$, respectively. We use the notion of "designed performance guarantee" in the same sense as in the previous section.

Lemma 3.2 *If $\lambda^* \leq \Lambda$, then algorithm Assign-R never fails. Thus, the load on a machine never exceeds 2Λ.*

procedure Assign-R$(\vec{p}, \vec{\ell}, \Lambda)$;

    /* Λ — current estimate of L^*.

    Let $S := \{i | \ell_i + p_i \leq 2\Lambda\}$;

    if $S = \emptyset$

      then $b :=$fail

      else begin

        $k := \min\{i | i \in S\}$;

        $\ell_k := \ell_k + p_k$;

        $b :=$success

      end;

    return$(\vec{\ell}, b)$.

end.

Figure 3: Algorithm Assign-R.

Proof: Assume Assign-R fails first on task j. Let r be the fastest machine whose load does not exceed L^*, ***i.e.*** $r = \max\{i | \ell_i(j-1) \leq L^*\}$. If there is no such machine, we set $r = 0$.

Obviously, $r \neq n$, otherwise j could have been assigned to the fastest machine n, since $\ell_n(j-1) + p_n(j) \leq L^* + L^* \leq 2\Lambda$. Define $\Gamma = \{i | i > r\}$, the set of ***overloaded*** machines. Since $r < n$, $\Gamma \neq \emptyset$. Denote by $\mathcal{S}_\rangle$ and by $\mathcal{S}^*_\rangle$ the set of jobs assigned to machine i by the on-line and the off-line algorithms, respectively. Since we are dealing with ***related*** machines, we have:

$$\sum_{\substack{i\in\Gamma,\\ s\in\mathcal{S}_\rangle}} p_n(s) = \sum_{\substack{i\in\Gamma,\\ s\in\mathcal{S}_\rangle}} \frac{p_n(s)}{p_i(s)} p_i(s) = \sum_{i\in\Gamma} \frac{v_i}{v_n} \sum_{s\in\mathcal{S}_\rangle} p_i(s) > \sum_{i\in\Gamma} \frac{v_i}{v_n} L^* \geq \sum_{i\in\Gamma} \frac{v_i}{v_n} \sum_{s\in\mathcal{S}^*\rangle} p_i(s) = \sum_{\substack{i\in\Gamma,\\ s\in\mathcal{S}^*_\rangle}} p_n(s) \qquad (7)$$

This implies that there exists a job $s \in \bigcup_{i\in\Gamma} \mathcal{S}_\rangle$, such that $s \notin \bigcup_{i\in\Gamma} \mathcal{S}^*_\rangle$, ***i.e.*** there exists a job assigned by the on-line algorithm to a machine $i \in \Gamma$, and assigned by the off-line algorithm to a slower machine $i' \notin \Gamma$.

By our assumptions, $p_{i'}(s) \leq L^* \leq \Lambda$. Since $r \geq i'$, machine r is at least as fast as machine i', and thus $p_r(s) \leq L^* \leq \Lambda$. Since job s was assigned before job j, $\ell_r(s-1) \leq \ell_r(j-1) \leq L^*$. But this means that the on-line algorithm should have placed job s on r or a slower machine instead of i, which is a contradiction. ∎

As we have mentioned above, the definition of the Assign-R algorithm facilitates a doubling approach to approximate Λ. More precisely, we start with $\Lambda = 0$. At the beginning of the first phase, Λ_1 is set to be equal to the load generated by the first job on the fastest machine for the job. At the beginning of a new phase $h > 1$, we set $\Lambda_h = 2\Lambda_{h-1}$. During a single phase, jobs are assigned independently of the jobs assigned in the previous phases. Phase h ends when Assign-R returns "fail". It is easy to see that this approach can increase the competitive factor by at most a factor of 4 (a factor of 2 due to the load in all the rest of the phases except the last, and another factor of 2 due to imprecise approximation of Λ). Since the designed performance guarantee of Assign-R is 2, we get:

Theorem 3.3 Algorithm Assign-R can be modified to achieve a competitive ratio of 8.

4 The Greedy Algorithm

The simple greedy machine load balancing algorithm due to Graham [19] gives a competitive ratio of 2 for the identical machines case and competitive ratio of $O(\log n)$ for the special case considered in [12]. It is natural to consider whether extensions of this algorithm can lead to small competitive ratios in the respectively more general cases of related and unrelated machines. In this section we show that, unfortunately, this is not the case. More precisely, we show that natural greedy approaches give $\Theta(n)$ competitive ratio for the unrelated machines case, and $\Theta(\log n)$ competitive ratio for the related machines case. This is in contrast to the $O(\log n)$ and 8 competitive ratios, respectively, produced by the algorithms presented in the previous sections.

We consider the following greedy algorithm: each job j is assigned upon arrival to the machine k that minimizes the resulting load, *i.e.,* the machine k that minimizes $\ell_k(j-1)+p_k(j)$.[1] Ties are broken by some arbitrary rule.

Lemma 4.1 The greedy algorithm has a competitive ratio no better than n for unrelated machines.

Proof: Consider a sequence of jobs such that job j has cost j on machine j, cost $1+\epsilon$ on machine $j-1$, and cost ∞ on all other machines (*i.e.,* $p_{j-1}(j) = 1+\epsilon, p_j(j) = j$, and $p_i(j) = \infty$ for all $i \neq j-1, j$). (To avoid distinguishing between the first job and all the other jobs, we refer to machine n also as machine 0.) Here ϵ is an arbitrarily small positive constant that is used to avoid ties. Clearly the optimal off-line algorithm can schedule all of these jobs with a maximum load of $1+\epsilon$ by assigning job j to machine $j-1$.

On the other hand, the greedy algorithm assigns job 1 to machine 1 for a resulting load of 1 (as opposed to $1+\epsilon$ on machine n or ∞ anywhere else). Similarly, when job 2 arrives, the greedy algorithm assigns it to machine 2 for a resulting load of 2, instead of assigning this job to machine 1, where it would have produced a load of $(2+\epsilon)$. Likewise, job 3 is assigned to machine 3 and so forth. A simple induction argument shows that job j is always assigned to machine j for a resulting load of j, giving a maximum load of n on machine n. The resulting performance ratio is $n/(1+\epsilon)$, which can be made arbitrarily close to n. ∎

Lemma 4.2 The competitive ratio of the greedy algorithm is at most n for unrelated machines.

Proof: Every job j has a minimum load $\min_i p_i(j)$ that can not be avoided by the optimal

[1] Note that assigning a job to the machine with the minimum load results in an algorithm with competitive ratio that is at least equal to the ratio of the fastest to slowest machine speeds.

off-line algorithm. If $\mathcal{S}_i^*$ is the set of jobs assigned to machine i by the off-line algorithm,

$$nL^* \geq \sum_i \ell_i^* = \sum_i \sum_{j \in \mathcal{S}_i^*} p_i(j) \geq \sum_j \min_i p_i(j).$$

On the other hand, we claim that the the maximum load resulting from the greedy algorithm will never exceed the sum of the minimum loads. Indeed, suppose that after assigning job $j-1$, we have $L(j-1) \leq \sum_{j' \leq j-1} \min_i p_i(j')$. When job j arrives there is some machine m that minimizes $p_m(j)$. The load on this machine is at most $L(j-1)$, and so if j is assigned to m the resulting load is at most $L(j-1) + p_m(j) \leq \sum_{j' \leq j} \min_i p_i(j')$. And thus, the actual assignment satisfies $L(j) \leq \sum_{j' \leq j} \min_i p_i(j')$. By induction, $L \leq \sum_j \min_i p_i(j) \leq nL^*$. ∎

Lemma 4.3 **The greedy algorithm has a competitive ratio $\Omega(\log n)$ for related machines.**

Proof: For simplicity, first we assume that whenever adding a job to two different machines will result in the same maximum load, the job is assigned to the faster machine. At the end of the proof we show how this assumption can be avoided.

Consider a collection of machines with speeds of the form 2^{-i} where $i \in \{0, 1, \ldots, k\}$ (the relation between k and n will become clear below). Let n_i be the number of machines with speed 2^{-i} and suppose $n_0 = 1$, $n_1 = 2$, and in general

$$n_i 2^{-i} = \sum_{j=0}^{i-1} n_j 2^{-j}. \tag{8}$$

These values are chosen so that the sum of the speeds of all machines with speed 2^{-i} is equal to the sum of the speeds of all the faster machines. Thus, a collection of jobs that would add 1 to the load of each machine with speed 2^{-i} could instead be assigned to add 1 to the loads of all the faster machines. The total number of machines $n = 1 + \frac{2}{3}(4^k - 1)$.

Now consider the following sequence of jobs. First we generate n_k jobs of size 2^{-k}, followed by n_{k-1} jobs of size $2^{-(k-1)}$, and so forth, until at last we generate a single job of size 1. For every machine with speed 2^{-j} there is a corresponding job of size 2^{-j}. By simply assigning each job to the corresponding machine, the off-line algorithm can schedule all the jobs with a resulting maximum load of 1.

However, we claim that the greedy algorithm assigns each group of jobs to machines that are "too fast". Assume, by induction, that before the jobs of size 2^{-i} are assigned, the load on machines with speed 2^{-j} is equal to $\min(k-i, k-j)$. (In the base case, when $i = k$, this condition simply corresponds to each machine having zero load.) By Equation 8, the greedy algorithm can assign the jobs of size 2^{-i} to all machines with speed $2^0, 2^{-1}, \ldots, 2^{-(i-1)}$, resulting in load of $k - i + 1$ on each one of these machines. If instead it assigns one of these jobs to a machine with speed 2^{-j}, where $j \geq i$, the resulting load will be $k - j + 2^{j-i} = (k - i) + 2^{j-i} - (j - i)$, which is at least $(k - i) + 1$ since $2^x - x \geq 1$ for all non-negative x. The greedy algorithm will therefore not assign any job of size 2^{-i} to a machine with speed 2^{-i} or slower, and the induction step follows.

Consequently, after all the jobs have been assigned, each machine with speed 2^{-j} has load $k-j$; the single machine with speed 1 has load $k = \Omega(\log n)$. Thus, under the simplifying assumption that the greedy algorithm always breaks ties in favor of the faster machine, the greedy algorithm is $\Omega(\log n)$-competitive.

Next we show how to avoid the above simplifying assumption. Before sending in the "large" jobs, we will send an "ϵ-job" of size $\epsilon_i 2^{-i}$ for each machine with speed 2^{-i}, giving it a load of ϵ_i. To avoid changing the greedy algorithm's choice of where to assign the large jobs, each ϵ_i must be less than 2^{-k}, the smallest possible difference between loads resulting from large jobs. To force ties to be broken in favor of faster machines we require that $\epsilon_j > \epsilon_i$ whenever $j > i$. Finally, to ensure that the ϵ-job intended for a machine is not placed on some faster machine, we generate the jobs for the faster machines first and require that $\epsilon_j < \epsilon_i + \epsilon_j 2^{-j}/2^{-i}$ for $j > i$. all of these conditions can be satisfied by choosing $\epsilon_i = 2^{-k-1}(1 + 2^{-2k+i})$. ∎

Lemma 4.4 **The greedy algorithm has a competitive ratio of $O(\log n)$ for related machines.**

Proof: Let L be the maximum load generated by the greedy algorithm and L^* be the maximum load generated by the optimal off-line algorithm. The structure of the proof is as follows. First, we show that the load on the fastest machines is at least $L - L^*$. Next, we show that if the load on all the machines with speed $\geq v$ is at least ℓ, then the load on all the machines with speed $\geq v/2$ is at least $\ell - 4L^*$. Repeated applications of this claim imply that the load on any machine that is no more than n times slower than the fastest machine, is at least $L - (1 + 4\lceil \log n \rceil)L^*$. Finally, we use an argument similar to the one used in the proof of Lemma 4.2 to show this condition can only hold if $L = O(\log n)L^*$.

First, consider the last job j assigned by the greedy algorithm to a machine i, causing the load of this machine to reach L. Since no job can add more than L^* to the load of any fastest machine, the fact that the new load on i is ℓ implies that the load on all the fastest machines is at least $L - L^*$.

Now suppose the load on all the machines with speed v or more is at least $\ell \geq 2L^*$. Consider the set of jobs that are responsible for the last $2L^*$ load increment on each one of these machines. Observe that at least one of these jobs (call this job j) has to be assigned by the offline algorithm to some machine i with speed less than v, and hence it can increase the load on i by most L^*. Since the speed of i is at most v, job j can increase the load on the machines with speeds above $v/2$ by at most $2L^*$. The fact that job j was assigned when the loads on all the machines with speed v and above was at least $\ell - 2L^*$ implies that the loads on all the machines with speed above $v/2$ is at least $\ell - 4L^*$.

Let v be the speed of the fastest machines. We have shown that all machines with speed v have load at least $L - L^*$. Iteratively applying the claim in the above paragraph shows that all machines with speed at least $v2^{-i}$ have load at least $L - L^* - 4iL^*$. Thus, every machine with speed at least v/n has load at least $L - (1 + 4\lceil \log n \rceil)L^*$.

Assume, for contradiction, that $L-(1+4\lceil\log n\rceil)L^* > 2L^*$. Recall that since we are in the related machines case, for each job j and any two machines i and i', we have $W(j)=p_i(j)v_i = p_{i'}(j)v_{i'}$, where $W(j)$ can be regarded as the weight of the job. Let I be the set of machines with speed less than $v_{\max}/n$. The total weight of jobs that can be assigned by the offline algorithm is bounded from above by

$$L^*\sum_{i=1}^{n} v_i \quad \le \quad nL^*\frac{v_{\max}}{n} + L^*\sum_{i\notin I} v_i \le 2L^*\sum_{i\notin I} v_i. \tag{9}$$

The assumption that the online algorithm causes load of more than $2L^*$ on all the machines not in I implies that the total weight of the jobs assigned by the online algorithm is greater than $2L^*\sum_{i\notin I} v_i$, which is a contradiction to 9. Thus, $L-(1+4\lceil\log n\rceil)L^* \le 2L^*$, which implies $L = O((\log ni)L^*)$. ∎

Acknowledgments

We are indebted to David Shmoys for many helpful discussions.

References

[1] Special issue on Asynchronous Transfer Mode. ***Int. Journal of Digital and Analog Cabled Systems***, 1(4), 1988.

[2] B. Awerbuch and Y. Azar. Local optimization of global objectives: Competitive distributed deadlock resulution and resource allocation. In ***Proc. 35th IEEE Symp. on Found. of Comp. Science***, pages 240–249, 1994.

[3] B. Awerbuch and Y. Azar. Competitive multicast routing. ***Wireless Networks***, 1:107–114, 1995.

[4] B. Awerbuch, Y. Azar, and A. Fiat. Packet routing via min-cost circuit routing. In ***4th Israeli Symposium on Theory of Computing and Systems***, pages 37–42, 1996.

[5] B. Awerbuch, Y. Azar, E. Grove, M. Kao, P. Krishnan, and J. Vitter. Load balancing in the l_p norm. In ***Proc. 36th IEEE Symp. on Found. of Comp. Science***, pages 383–391, 1995.

[6] B. Awerbuch, Y. Azar, and S. Plotkin. Throughput competitive on-line routing. In ***Proc. 34th IEEE Annual Symposium on Foundations of Computer Science***, pages 32–40, November 1993.

[7] B. Awerbuch, Y. Azar, S. Plotkin, and O. Waarts. Competitive routing of virtual circuits with unknown duration. In ***Proc. 5th ACM-SIAM Symposium on Discrete Algorithms***, pages 321–327, January 1994.

[8] B. Awerbuch, R. Gawlick, T. Leighton, and Y. Rabani. On-line admission control and circuit routing for high-performance computing and communication. In *Proc. 35th IEEE Annual Symposium on Foundations of Computer Science*, pages 412–423, November 1994.

[9] B. Awerbuch, Y.Bartal, A. Fiat, and A. Rosen. Competitive non-preemptive call control. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[10] Y. Azar, A. Broder, and A. Karlin. On-line load balancing. In *Proc. 33rd IEEE Annual Symposium on Foundations of Computer Science*, pages 218–225, 1992.

[11] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. On-line load balancing of temporary tasks. In *Proc. Workshop on Algorithms and Data Structures*, pages 119–130, August 1993.

[12] Y. Azar, J. Naor, and R. Rom. The competitiveness of on-line assignment. In *Proc. 3rd ACM-SIAM Symposium on Discrete Algorithms*, pages 203–210, 1992.

[13] Y. Bartal, A. Fiat, H. Karloff, and R. Vohra. New algorithms for an ancient scheduling problem. In *Proc. 24th Annual ACM Symposium on Theory of Computing*, 1992.

[14] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. *J. ACM*, (39):745–763, 1992.

[15] J. Garay, I. Gopal, S. Kutten, Y. Mansour, and M. Yung. Efficient on-line call control algorithms. In *Proc. of 2nd Annual Israel Conference on Theory of Computing and Systems*, 1993.

[16] J.A. Garay and I.S. Gopal. Call preemption in communication networks. In *Proc. INFOCOM '92*, volume 44, pages 1043–1050, Florence, Italy, 1992.

[17] R. Gawlick, C. Kalmanek, and K. Ramakrishnan. On-line permanent virtual circuit routing. In *Proceedings of IEEE Infocom*, April 1995.

[18] R. Gawlick, A. Kamath, S. Plotkin, and K. Ramakrishnan. Routing and admission control in general topology networks. Technical Report STAN-CS-TR-95-1548, Stanford University, 1995.

[19] R.L. Graham. Bounds for certain multiprocessing anomalies. *Bell System Technical Journal*, 45:1563–1581, 1966.

[20] R.L. Graham, E.L. Lawler, J.K Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[21] A. Kamath, O. Palmon, and S. Plotkin. Routing and admission control in general topology networks with poisson arrivals. In *Proc. 7th ACM-SIAM Symposium on Discrete Algorithms*, pages 269–278, 1996.

[22] D. Karger, S. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. Unpublished manuscript, 1993.

[23] D. Karger and S. Plotkin. Adding multiple cost constraints to combinatorial optimization problems, with applications to multicommodity flows. In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 18–25, May 1995.

[24] A.R. Karlin, M.S. Manasse, L.Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 1(3):70–119, 1988.

[25] R. Karp, U. Vazirani, and V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 352–358, 1990.

[26] P. Klein, S. Plotkin, C. Stein, and É. Tardos. Faster approximation algorithms for the unit capacity concurrent flow problem with applications to routing and finding sparse cuts. Technical Report 961, School of Operations Research and Industrial Engineering, Cornell University, 1991.

[27] J. Kleinberg and É. Tardos. Disjoint path in densely embedded graphs. In *Proc. 36th IEEE Annual Symposium on Foundations of Computer Science*, 1995.

[28] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problem. In *Proc. 23th ACM Symposium on the Theory of Computing*, pages 101–111, May 1991.

[29] Y. Ma and S. Plotkin. Improved lower bounds for load balancing of tasks with unknown duration. Unpublished manuscript, 1995.

[30] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for online problems. In *Proc. 20th Annual ACM Symposium on Theory of Computing*, pages 322–332, 1988.

[31] S. Phillips and J. Westbrook. Online load balancing and network flow. In *Proc. 25th Annual ACM Symposium on Theory of Computing*, pages 402–411, 1993.

[32] S. Plotkin. Competitive routing in ATM networks. *IEEE J. Selected Areas in Comm.*, pages 1128–1136, August 1995. Special issue on Advances in the Fundamentals of Networking.

[33] S. Plotkin, D. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. In *Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science*, pages 495–504, October 1991.

[34] F. Shahrokhi and D. Matula. The maximum concurrent flow problem. *J. Assoc. Comput. Mach.*, 37:318–334, 1990.

[35] D. Shmoys, J. Wein, and D.P. Williamson. Scheduling parallel machines on-line. In *Proc. 32nd IEEE Annual Symposium on Foundations of Computer Science*, pages 131–140, 1991.

[36] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Comm. ACM*, 28(2):202–208, 1985.

[37] H. Takahashi and A. Matsuyama. An approximate solution for the steiner problem in graphs. *Math. Japonica*, 24, 1980.