CS369 - Online Algorithms
Lecture Notes for 2012-2013 Academic Year

April 2013

Contents

1 Online Algorithms

1.1 Introduction

An online problem is one where not all the input is known at the beginning. Rather, the input is presented in stages, and the algorithm needs to process this input as it is received. As the algorithm does not know the rest of the input, it may not be able to make optimum decisions. Online algorithms were first studied in [14],[23] in the context of bin-packing problems. They gained importance after the path breaking work in [22].

1.2 The Competitive Ratio: Renting vs. Buying Skis

A simple example of an online problem is the ski-rental problem. A skier can either rent skis for a day or buy them once and for all. Buying a pair of skis costs k times as much as renting it. Also, skis once bought cannot be returned. Whenever the skier goes skiing, she does not know whether she will ski again or how many times. This makes the problem an online one.

Consider the following scenario. A malicious Weather God lets the skier ski as long as she is renting skis, but does not let her go on any skiing trips after she has bought them. Suppose the skier decides to rent skis on the first k' trips and buys them on the next trip. The total cost incurred by the skier is $k' + k$. But the optimum cost is k if $k' + 1 \geq k$ and $k' + 1$ otherwise. Therefore, the skier ends up paying a factor of $(k' + k)/\min(k, k' + 1)$ as much as the optimum. Choosing $k' = k - 1$, this factor equals $2 - 1/k$.

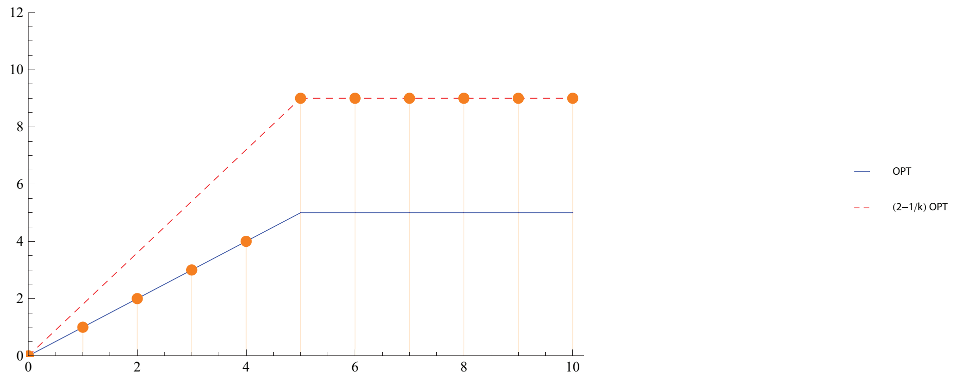


Figure 1: A plot for $k = 5$. Our algorithm starts with points on OPT, but must eventually contain a point on the dotted line. This tells us the minimum multiple of OPT we can achieve.

This factor is called the *Competitive Ratio* of the algorithm. Note that while calculating this factor, we used the worst possible input sequence. More formally, an online algorithm A is *α-competitive* if for all input sequences σ,

$$C_A(\sigma) \leq \alpha \cdot C_{OPT}(\sigma) + \delta$$

where C_A is A's cost function, C_{OPT} is the optimum's cost function, and δ is some constant. We can look upon the competitive ratio as the inherent cost of not knowing the future.

[0]Edited by Lucas Garron.

For the ski problem, $2 - 1/k$ is both a lower and an upper bound on the best competitive ratio – no online algorithm can do better than this, and there is an online algorithm that achieves this.

1.3 Paging

Paging [22] is a more realistic example of an online problem. We have some number of pages, and a cache of k pages. Each time a program requests a page, we need to ensure that it is in the cache. We incur zero cost if the page is already in the cache. Otherwise, we need to fetch the page from a secondary store and put it in the cache, kicking out one of the pages already in the cache. This operation costs one unit. The online decision we need to make is *"which page to kick out when a page fault occurs"*. The initial contents of the cache are the same for all algorithms.

Some online strategies are:

LIFO (Last In First Out)

FIFO (First In First Out)

LRU (Least Recently Used - the page that has not been requested for the longest time is kicked out)

LFU (Least Frequently Used)

We now show that k is a lower bound on the competitive ratio of any deterministic online strategy for the paging problem. This lower bound can actually be matched by both **LRU** and **FIFO**.

1.3.1 Upper Bound using LRU

Theorem 1.1 **LRU** *is k-competitive for the online paging problem [22].*

Proof Sketch: Define a *phase* as a sequence of requests during which **LRU** faults exactly k times. All we need to show is that during each phase, the optimal algorithm (OPT) must fault at least once.

Case 1 **LRU** *faults at least twice on some page σ_i during the phase.* In this case, σ_i must have been brought in once and then kicked out again during the phase. When σ_i is brought in, it becomes the most recently used page. When it is later kicked out, σ_i must be the least recently used page. Therefore, each of the other $k-1$ pages in the cache at this time must have been requested at least once since σ_i was brought in. Counting σ_i and the request that causes σ_i to be evicted, at least $k+1$ distinct pages have been requested in this phase. Therefore, OPT must fault at least once.

Case 2 **LRU** *faults on k different pages.* If this is the first phase, then OPT must fault at least once since OPT and **LRU** start off with the same pages in their cache. Else, let σ_i be the last page requested in the previous phase. This page has to be in both **LRU**'s and OPT's cache at the start of the phase. If **LRU** does not fault on σ_i, then the k pages on which **LRU** faults cannot all be present in OPT's cache at the beginning of the phase. Therefore, OPT must fault at least once. The case when **LRU** does fault on σ_i is very similar to Case 1.

1.3.2 Lower Bound using LFD

Let A be an algorithm for the paging problem. Consider a malicious program that uses only $k+1$ pages. At each step, the program requests the one page that is not in A's cache. So the cost incurred by A is $|\sigma|$.

LFD (Longest Forward Distance) is a strategy that kicks out the page that is going to be requested furthest in the future [1]. When **LFD** kicks out a page σ_i, the next page fault can occur only when this page is requested again, since there are only $k+1$ pages. But by the definition of **LFD**, each of the $k-1$ pages that was not kicked out at the first page fault must be requested before σ_i is requested again. Thus, page faults are at least a distance k away and the cost of **LFD** is at most $(|\sigma|-1)/k+1$. Therefore A can be no more than k-competitive. This is not a good competitive ratio at all, since cache sizes can typically be very large. However, **LRU** performs very well in practice.

1.4 Oblivious Adversary

For randomized online algorithms, we need to define our adversary carefully. An *oblivious adversary* is one that does not know the coin tosses of the online algorithm – it knows the "source code" of the online algorithm, without access to its random source during execution. An online algorithm A is α-competitive against oblivious adversaries if for all inputs σ,

$$E\Big[C_A(\sigma)\Big] \leq \alpha \cdot C_{OPT}(\sigma) + \delta \ ,$$

where δ is a fixed constant, and the expectation is taken over the random coin tosses.

1.5 Marking Algorithm: MARK

We now present **MARK**, the Marking algorithm for randomized paging and show that it is $O(\log k)$-competitive against oblivious adversaries. The result is due to [12], where they also give a (nearly) matching lower bound.

When a request for a page p comes in, **MARK** does the following :

- If the page p is not in the cache
  - If all pages are marked, unmark them all.
  - Randomly choose an unmarked page for eviction.
- Mark p.

1.5.1 Deterministic MARK

Note that **LRU is a deterministic version of MARK**, where one uses a specific deterministic rule to choose which one of the unmarked pages to evict [2]. Let us consider an intermediate version of **MARK**

[1]**LFD** is actually an optimum strategy for paging - unfortunately it assumes knowledge of future requests.

[2]In practice, it is useful to implement **LRU** more like **MARK** by ignoring the difference between sufficiently old /unmarked slots.

where we choose an unmarked page for eviction deterministically.

We will consider a *phase* to be the time between two unmarking resets by deterministic **MARK**. During one phase, we bring in k new pages.

To be a little more general, suppose OPT has a cache of size h. OPT can prepare $h-1$ slots (1 slot must be taken by the previous request) before the phase, and must fault on the remaining $k-(h-1)$. Thus, the competitive ratio is $\frac{k}{k-(h-1)}$. For $k=2h$, this is 2-competitive.

However, when $k=h$, we get competitive ratio of k for **MARK** using a deterministic rule. We need randomization to do better.

1.5.2 Randomized MARK

Let a *phase* be a sequence during which k different pages are requested. At the beginning of each phase, all pages in **MARK**'s cache are marked. Since a page requested during a phase is never evicted again during the phase, we need to consider only the first time a page is requested within a phase. Let S_i be the set of pages in **MARK**'s memory at the beginning of phase i. A request p during a phase is "old" if $p \in S_i$ and "new" otherwise.

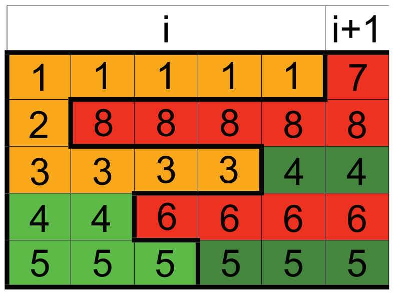


Figure 2: A $k=5$ example of a phase i with $S_i = \{1,2,3,4,5\}$, $S_{i+1} = \{4,5,6,7,8\}$, and $l_i = |S_{i+1} - S_i| = 3$. Page 4, is accidentally kicked out and brought back in, resulting in 1 more fault than the minimum, l_i.

在第i个阶段的新的被叫

Let l_i be the number of new requests during phase i. Each *new* request must cost us one unit, so we need to calculate the costs of requests for *old* pages during the phase (i.e. the number of times we accidentally kick out an old request and have to bring it back in). Consider the point during phase i where we've served s requests for old pages, and are we are serving request number $s+1$ for an old page. The s previously requested old pages are already in memory, in addition to (at most) l_i new pages. So at most l_i old pages have been kicked out, and they have been chosen randomly from the $(k-s)$ old pages not yet requested. The probability that p was one of these is (at most) $l_i/(k-s)$. Summing over s going from 0 to $k - l_i - 1$, the expected total cost of all the old requests in a phase is at most

因为新来了li个被叫所以最多有li个old被踢出

$$l_i \left(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{k-(k-l_i-1)} \right)$$

调和数

which is no more than $l_i(H_k - 1)$, where H_k is the k th harmonic number. Adding the cost of all new requests, the expected total cost of **MARK** is bounded by $\sum_i l_i \cdot H_k$.

Lemma 1.2 *The cost of OPT on an input sequence is at least* $\frac{1}{2}\sum_i l_i$.

The main idea is to show that if an offline algorithm does not fault on many new requests during phase i then it must evict a lot of pages in the previous and following phases. Even though it is variable, we can take the performance of **MARK** as a baseline and compare OPT against it.

Let ϕ_i be the number of pages in S_i (**MARK**'s cache) that are not in OPT's cache at the beginning of phase i. Then the following two conditions hold:

$\underline{C_{OPT}(phase_i) \geq l_i - \phi_i}$. During phase i, l_i of the pages requested are not in S_i. But OPT's cache differs from S_i in at most ϕ_i pages. Therefore, OPT must pay $l_i - \phi_i$.

$\underline{C_{OPT}(phase_i) \geq \phi_{i+1}}$. Each of the k pages requested in the ith phase is present in S_{i+1}. Also each of these pages was present in OPT's cache at some time during phase i. But OPT's cache at the start of the next phase does not have ϕ_{i+1} of these pages - so there must have been at least ϕ_{i+1} evictions by OPT during the ith phase.

Adding the two together over all phases, we get $C_{OPT} \geq 1/2\Big(\sum_i (l_i) - \phi_0 + \phi_n\Big)$, where n is the numbers of phases. Since ϕ_0 is 0, this completes the proof of the lemma.

We can now conclude that **MARK** is $2H_k$ competitive against oblivious adversaries. And since $H_k = O(\log k)$, the result follows. Notice that if the adversary were not oblivious, it could have chosen a freshly evicted page for all old requests.

2 Online Steiner Tree Algorithms

We will discuss online and off-line approximation algorithms for calculating Steiner trees on metric spaces [2]. Using Eulerian walks, we will show a *log*-competitive on-line algorithm, and a 平面的 planar "jellyfish" graph show *log*-competitive is also the lower bound. In particular, we will use $Q-$disks to discretize the plane and express by a node covering argument that either (1) our existing tree $\mathcal{T}$ costs a lot, or (2) it will cost a lot to add nodes to it.

Def. 1 (Steiner Tree Problem) *Given a graph $G = (V, E)$ and a subset $U \subseteq V$ ("red" nodes), find the tree of A of minimal cost that connects all nodes of U.*

Connecting these nodes optimally is an NP-complete problem, with many good approximations for undirected graphs and has a practical application in handling multicast routing.

2.1 Offline 2-Approximation

Connecting "red" Nodes

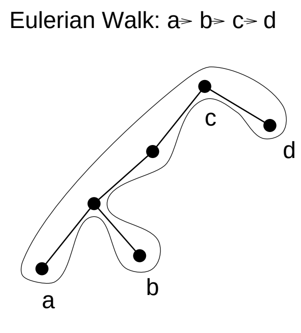
Algorithm 2.1 *The following steps describe an approximate offline algorithm for Steiner tree.*

- Generate a complete graph on "red" nodes (edge cost equal to the minimum cost path on the original graph, G). Let us call this complete graph K_G.
- Calculate the MST of K_G; the sum of edge costs is the tree's weight *W(MST)*.
- Reinterpret this MST on the original graph. Note that this may require pruning redundant edges. Thus the weight of the resulting tree, $W(result)$, is at most $W(MST)$.

Lemma 2.2 *Algorithm 2.1 gives a 2-approximation, due to: $W(result) \leq W(MST) \leq 2OPT$.*

Consider the Eulerian walk on the optimal Steiner tree. The walk visits each edge exactly twice, or exactly once from either direction. The cost of this walk, W_e, is simply $2OPT$.
If we take shortcuts in this walk by replacing a path between each pair of consecutive red nodes a, b by their shortest path, we do not increase the cost (because $dist(a, b)$ is no greater than any path from a to b); by consecutive red nodes, we mean the two nodes a and b are visited one after another in the walk. Using such shortcuts, we could transform the Eulerian walk into a cycle on all red nodes, whose cost is no greater than W_e. Now if we remove any edge from this cycle, it becomes a path that connects all red nodes (which is a spanning tree), whose cost is at least $W(MST)$. Thus we have $W_e \geq W_{cycle} \geq W_{path} \geq W(MST)$, and in particular $W_e \geq W(MST)$.
The resulting tree of Algorithm 2.1 has the weight $W(result)$ which is by definition no greater than $W(MST)$, because the resulting tree could only decrease the weight by pruning unnecessary/redundant edges in the MST. Altogether, we have the following:

$$W(result) \leq W(MST) \leq W_e = 2OPT$$



[2]Edited by Hooyeon Lee.

2.2 Online $\log n$-Approximation

Now, assume that nodes of the graph may become "red" on-line. The problem becomes one of maintaining the Steiner tree as more "red" nodes are revealed. Note that the underlying graph G is given to begin with, while the set of red nodes, U, is revealed one node by one node.

Algorithm 2.3 *Suppose we have red nodes U' and we want to connect in a new red node $u' \notin U'$. Then, we find $v' \in U'$ s.t. $dist(u', v')$ is minimal and add all edges connecting u' and v' into our set. Note that v' is a red node as well (since $v' \in U'$). We denote the cost of adding u' as $c(u')$, which is the weight of newly added edges on the path between u' and v'.*

We show that the resulting Steiner tree of this algorithm is log-competitive with the optimal solution.

Lemma 2.4 *For any fixed $L \geq 0$, let $S_L \subseteq U$ be the set of "red" nodes whose cost is at least L, or formally $S_L = \{v \in U : c(v) \geq L\}$. Then we have $|S_L| \leq 2OPT/L$ where OPT is the weight of the optimal Steiner tree that connects all nodes in U. Equivalently, we also have $L \leq 2OPT|S_L|$.*

Proof: Since each node in S_L incurred the cost at least L when it was added to the Steiner tree (by Algorithm 2.3), the distance between any two nodes in S_L is at least L. Thus, any Hamiltonian cycle on S_L costs at least $|S_L|L$. Now consider the optimal Steiner tree, $\mathcal{T}_L$, on S_L whose cost is denoted by OPT_L (to distinguish from OPT). Note that $OPT_L \leq OPT$ (since OPT_L is the cost of the optimal Steiner tree on a subset of U, namely S_L). If we take the Eulerian walk on $\mathcal{T}_L$, the cost of this walk is exactly $2OPT_L$ (since each edge is visited twice). Using the same technique from our previous lemma, we can turn this Eulerian walk into a Hamiltonian cycle of nodes in S_L such that its net cost is no greater than $2OPT_L$. That is, if W_H is the cost of this Hamiltonian cycle constructed out of the Eulerian walk on $\mathcal{T}_L$, then we have $W_H \leq 2OPT_L$. We earlier showed that any Hamiltonian cycle on S_L costs at least $|S|L$, so we have $|S_L|L \leq W_H \leq 2OPT_L \leq 2OPT$. This proves the desired inequality $|S_L| \leq 2OPT/L$. ▮

Using this lemma, we can easily show that the online algorithm is $O(\log n)$-competitive with the optimal solution (where $n = |U|$). Let us denote $L_i = c(v_i)$ where v_i's are red nodes in U that are ordered by their costs, $c(v_i)$, from largest to smallest. The net cost of the Steiner tree given by our online algorithm is simply $\sum_{i=1}^{|U|} L_i$. For each L_i, we know from the lemma above that $L_i \leq 2OPT/|S_{L_i}| \leq 2OPT/i$. Summing all these up over i, we get that $\sum_{i=1}^{|U|} L_i \leq O(OPT \log |U|)$.

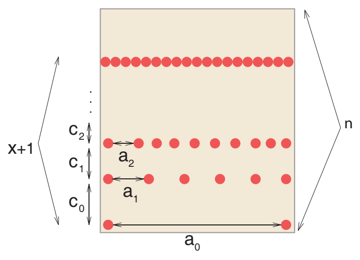
2.3 Lower Bound

The lower bound is due to [2]. To prove the lower bound, we will consider a "square-grid" graph with $(n+1) \times (n+1)$ nodes in the Euclidean (L_2) plane. The graph is complete, with weights on edges equal to the Euclidean distance between the endpoints. The adversary constructs layers of "red" nodes and presents them to the online algorithm layer-by-layer. The lowest layer consists of 2 red nodes. Second layer has more nodes, third layer more than the second one, and so on such that the highest layer consists of $(n+1)$ red nodes.

The adversary is going to reveal a layer of red nodes from lowest to highest such that total $x+1$ layers of red nodes will be revealed to the online algorithm. More precisely, we consider the following setting:

- Construct a grid graph with $(n+1)\times(n+1)$ nodes. This is the underlying graph. The adversary will reveal total $x+1$ layers of "red" nodes among $(n+1)^2$ nodes in the graph. n is parameterized by x with $n = x^{2x}$.
- For layer i (with $0 \le i \le x$), let a_i be the horizontal distance between red nodes in layer i. We set $a_0 = n$ and $a_i = a_{i-1}/x^2 = a_0/x^{2i}$ for all $i \ge 1$.
- Let c_i be the vertical distance between two adjacent layers i and $i+1$ (with $0 \le i < x$). c_i is parameterized by an integer k_i such that $c_i = k_i a_{i+1}$ where $1 \le k_i \le x$ (k_i is chosen by the adversary, based on behavior of the online algorithm). Notice that $1 \le k_i \le x \Rightarrow a_{i+1} \le c_i \le a_i/x$.

Construction for lower bound proof



Several helpful observations:

- Each layer has total $(n/a_i)+1$ red nodes. In particular, layer 0 has two red nodes and layer x has $n+1$ red nodes.
- Given x, the adversary's only freedom is to choose k_i in this construction. Choosing larger values of k_i, the adversary "stretches out" the set of red nodes. Conversely, using small values of k_i for each layer, the adversary reduces the vertical dimension of the region that has red nodes.
- The position of the top red layer in the underlying graph is determined by the choices of k-values for all the layers.

Lemma 2.5 *The optimal Steiner tree can connect all red nodes in this example with $O(n)$ cost.*

Proof: We begin by describing simple Steiner tree that achieves $O(n)$ to connect all red nodes in this example. First, we connect all $(n+1)$ red nodes in the top layer (or layer x), which costs n. Then, we connect layer $i+1$ and i (for decreasing i from $x-1$ to 0) such that the red nodes in the lower layer i are "hooked" up to the red nodes in the upper layer i. We use this scheme because c_i (the distance between two layers) is always smaller than a_i (recall that $c_i \le a_i/x$), which means we are better off if we connect red nodes in layer i vertically to red nodes in layer $i+1$, instead of connecting them horizontally. Since layer i contains n/a_i+1 red nodes, hooks between layer $i+1$ and i cost exactly $c_i(n/a_i+1)$. The total cost of this Steiner tree is given by:

$$\begin{aligned}
n + \sum_{i=0}^{x-1} c_i(n/a_i+1) &= n\left(1+\sum_{i=0}^{x-1}(c_i/a_i + c_i/n)\right)\\
&\le n\left(1+\sum_{i=0}^{x-1} 2(c_i/a_i)\right)\\
&\le n\left(1+\sum_{i=0}^{x-1} 2/x\right) = 3n
\end{aligned}$$

The first inequality is due to $a_i \le n$, and the second inequality is due to $c_i/a_i \le 1/x$. This Steiner tree is intuitively optimal (while we are not going to formally prove that it is optimal), thus we know that the optimal Steiner tree can achieve $O(n)$. ▌

Now suppose we (the adversary) have the freedom of choosing integers k_i with $1 \leq k_i \leq x$ (for i with $0 \leq i \leq x-1$), from k_0 to k_{x-1}, one by one. The following lemma says that we can always choose the right value for k_i such that the online algorithm cannot be better than *log*-competitive.

Lemma 2.6 *Let $W(\mathcal{T}_i)$ be the weight (or cost) of the current Steiner tree of the online algorithm before the adversary reveals layer $i+1$. We (the adversary) can choose k_i such that either (1) layer i costs at least $n/8$, or (2) the total cost, $W(\mathcal{T}_i)$, is at least $nx/8$.*

Proof: We focus out attention to the case where the adversary has revealed the first i layers (from layer 0 to layer $i-1$), and is trying to choose the right value for k_i. The online algorithm has constructed its Steiner tree that connects red nodes up to layer $i-1$.

Recall that $c_i = k_i a_{i+1}$ with $1 \leq k_i \leq x$. We have x possible values for k_i, which determines how far layer $i+1$ is placed from layer i. Since there are x possible locations, we can imagine this as a grid of x candidate layers (vertically distanced by a_{i+1} between layers). Since layer $i+1$ would contain $(n/a_{i+1}+1)$ red nodes, the grid of our interest consists of $x \times (n/a_{i+1}+1)$ candidate nodes. For each candidate node in the grid, consider a P-disk of radius $a_{i+1}/2$ and a Q-disk of radius $a_{i+1}/4$, both of which are centered at the candidate node. That is, each candidate node has P-disk at its center inside which a Q-disk is placed. Note that all P-disks are disjoint because each candidate node is at least a_{i+1} apart from its horizontally/vertically adjacent nodes.

For these x candidate layers of Q-disks, let Q_l be the set of Q-disks (on layer l) that do not have any points in the existing tree of the online algorithm, $\mathcal{T}_i$. Let $r_l = |Q_l|$ and $r = \sum_{l=1}^{x} r_l$. Our intuition is that if r is small, then the cost of adding layer $i+1$ to the existing tree $\mathcal{T}_i$ should be relatively small, but the cost of this preparation should be too large. We consider the following two cases.

Case 1) $r > \frac{x}{2}(n/a_{i+1}+1)$

In this case, there must exist some l such that $r_l \geq (n/a_{i+1}+1)/2$ (if no such l exists, then r cannot be greater than $\frac{x}{2}(n/a_{i+1}+1)$). The adversary will choose this layer l, and the cost of adding all red nodes in this layer would cost the online algorithm at least $(a_{i+1}/4) \cdot (n/a_{i+1}+1)/2 \geq n/8$.

Case 2) $r \leq \frac{x}{2}(n/a_{i+1}+1)$

In this case, the current tree $\mathcal{T}_i$ must touch at least $\frac{x}{2}(n/a_{i+1}+1)$ Q-disks (because the total number of Q-disks is $x(n/a_{i+1}+1)$). To touch each Q-disk, the tree must penetrate the surrounding "halo" of the Q-disk (which is the area between the Q-disk and its co-centered P-disk), which would cost the tree at least $a_{i+1}/4$ (the difference between the radii of the two disks). Hence, the total cost of current tree is at least $\frac{x}{2}(n/a_{i+1}+1) \cdot a_{i+1}/4 \geq nx/8$. ▮

Due to the lemma, we know either (1) that all x layers would cost the online algorithm at least $n/8$ each, or (2) that the total cost of the online algorithm exceeds $nx/8$ at some point. In the case of (1), the total cost of the online algorithm is at least $nx/8$ because there are x layers (thus, $O(nx)$). In the case of (2), the adversary can stop at that point because the online algorithm is already $O(nx)$. Since $O(nx) = O(n \log n)$, this completes our proof for a lower-bound for online Steiner tree algorithm.

Note: It is possible to extend the result to oblivious adversaries by observing that we can transform both cases into a form with an expected value. (In the first case we choose a random l, and use the property of linearity of expectation.)

3 Online Load Balancing

In this chapter, we will consider the problem of *Online 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 an 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.

It is important to note that *load* in this context is not equivalent to termination time. In a model based on termination time, the release time of jobs becomes important. The model used here can be understood by thinking of jobs such as services (eg. DNS server, Web server, etc) that add some amount of load to a machine, but have no expected termination time.

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 $l_i(j)$ denote the load on machine i after we have already assigned jobs 1 through j:

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

Consider a sequence of jobs defined by $\sigma = (\vec{p}(1), \vec{p}(2), \ldots \vec{p}(k))$. Denote by $l_i^*(j)$ 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 on-line algorithms is to minimize $L^*(k) = \max_i l_i^*(k)$ and $L(k) = max_i l_i(k)$, 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).

The load balancing problems can be categorized into three classes according to the properties of the load vector.

Identical machines: All the coordinates of a load vector are the same. In other words, for a particular job, all machines would incur the same load as a result of running that job.

$$\forall i, i', j : p_i(j) = p_{i'}(j)$$

Related machines: 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.

$$\forall i, i', j'j' : \frac{p_i(j)}{p_{i'}(j)} = \frac{p_i(j')}{p_{i'}(j')} = v_{i'}/v_i$$

Unrelated machines: This includes all other cases. This class could include, for example, load balancing problems in which some machines have a advantage (beyond raw speed) in executing particular jobs, due to factors such as architectural differences, software differences, etc. A special case of the unrelated machines class of problems is the $1/\infty$ model, in which some machines get 1 unit of load if assigned a job, while others cannot execute the job at all.

3.1 Identical Machines

We use a greedy algorithm for identical machines [13] [3].

[2]Edited by Alex Churchill and Brett Wines.

[3]Strictly less than 2 competitive ratio is achievable - see [1]

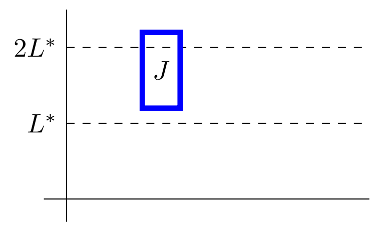


Figure 3: Assume some job J takes us above $2L^*$

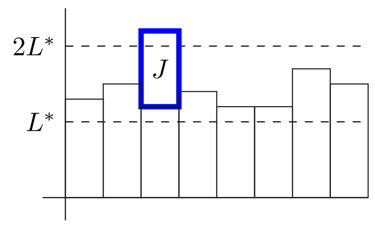


Figure 4: Then all machines must have been loaded above L^*

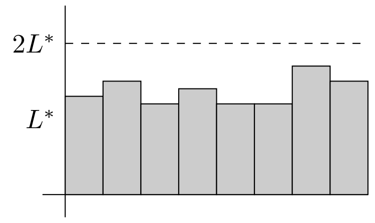


Figure 5: But this means the total load must be greater than nL^*, which is a contradiction

Algorithm 3.1 *put new job on least loaded machine*

Lemma 3.2 *Greedy algorithm is 2-competitive*

- **Consider** a machine i, which when receiving a new job, has $L_i > 2L^*$.
- $\Rightarrow$ all current loads $> L^*$ (as $p_i(s) \leq L^*, \forall i \forall s$)
- $\Rightarrow \sum_i L_i > nL^* \Rightarrow$ contradiction! (as the total load cannot be redistributed by the optimal offline algorithm so that each machine has load $\leq L^*$)
- $\Rightarrow L_i < 2L^*$

To be precise, the load Q of our busiest machine is such that $n(Q-l)+l \leq nL^*$. Therefore, $Q \leq l(n-1)/n + L^* \Rightarrow Q \leq (1+(n-1)/n)L^* = (2-1/n)L^*$. This demonstrates there exists a bound strictly less than two. Proofs of the $(2-\epsilon)$ bounds for a small constant ϵ have appeared in [10, 15, 1].

3.2 Related Machines

This section mostly follows discussion in [3].

The greedy algorithm The simple greedy load balancing algorithm due to Graham [13] achieves a competitive ratio of $2-\frac{1}{n}$ for the identical machines case. It is easy to see that applying it "as is" to related machines case is not going to work. In particular, if there is an extremely slow machine with very low current load, we might decide to use it for the next job, which in turn might cause a very significant increase in load on this machine, arbitrary larger than L^*.

A straightforward modification to avoid the above case is to assign every job to the machine that *will execute it with the lowest resulting load.* In other words, when assigning job j, instead of looking for machine i that minimizes $l_i(j-1)$, look for i that minimizes $l_i(j-1)+p_i(j)$. In this section we will refer to the resulting algorithm as "greedy".

Algorithm 3.3 *Put new job on machine which minimizes max load after assigning the job*

First we prove a lower bound on the greedy algorithm:

Theorem 3.4 *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 will 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

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

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 1, 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 $\epsilon_i = 2^{-k-1}(1 + 2^{-2k+i})$. ∎

Theorem 3.5 *The greedy algorithm has a competitive ratio* $O(\log n)$ *for related machines.*

Proof: Let L be the maximum load achieved by the greedy algorithm and let L^* be the maximum load achieved by the optimal off-line algorithm.

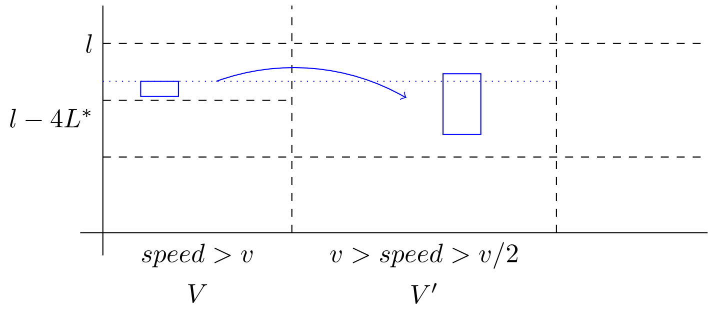


Figure 6: There is some job (the blue block) which is loaded in our algorithm on a machine in V (faster than v) which in the optimal is loaded on a machine slower than V. Therefore, on a machine in V', it has load at most $2L^*$. But the load of any machine in V' plus $2L^*$ must be greater than the load on the machine our algorithm selected in V, which is at least $l - 2L^*$. Therefore, the load on any machine in V' is at least $l - 2L^* - 2L^* = l - 4L^*$.

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 $l - L^*$. Otherwise, the greedy algorithm would have assigned job j to a faster machine to achieve a load lower than l.

Intuitively, what we are going to show is that if the load on the fastest machines is very large, then the load on all of the machines is, although smaller, but still large. Then, to bound load on the fastest machines, we will use the fact that it is not possible to have uniformly large (above L^*) load on all of the machines.

Lemma 3.6 *If the load on all machines with speed $\geq v$ is $\geq l$, then the load on all machines with speed $\geq v/2$ is $\geq l - 4L^*$.*

Proof: Now suppose that the load on all machines with speed v or more is at least $l \geq 2L^*$. Consider the set of jobs that are responsible for the last $2L^*$ load increment on each of these machines. Observe that at least one of these jobs (call this job j) has to be assigned by the off-line algorithm to some machine i with speed less than v, and hence it can increase the load on i by at most L^*. Since the speed of i is at most v, job j can increase the load on all 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 were at least $l - 2L^*$ implies that the loads on all the machines with speed above $v/2$ were at least $l - 4L^*$. See figure 6. ∎

Let v_{max} be the speed of the fastest machines. We have shown that all machines with speed v_{max} have load at least $L - L^*$. Iteratively applying the above claim shows that all machines with speed $v_{max}/2^i$ have load at least $L - L^* - 4iL^*$. Thus every machine with speed at least v_{max}/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 off-line algorithm is bounded from above by

$$L^* \sum_{i=1}^{n} v_i \leq nL^* \frac{v_{max}}{n} + L^* \sum_{i \notin I} v_i \leq 2L^* \sum_{i \notin I} v_i$$

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

Achieving a constant competitive ratio Let us assume we know the optimal offline load L^* (we shall explain later as to how do we deal with the fact that we do not know this). Consider an algorithm with the following allocation strategy: when placing job j, consider $S = \{i | l_i(j-1) + p_i(j) \leq 2L^*\}$. Put j on the slowest machine $i \in S$.

Lemma 3.7 *At every step of the algorithm, $S \neq \emptyset$.*

Proof: Assume, for contradiction, that $S = \emptyset$. (We will use, for notational purposes, that machines are ordered by increasing speed. i.e. For machines a and b where $1 \leq a \leq b \leq n$ we have $v_a \leq v_b$.)

Let r be the fastest machine whose load does not exceed L^*, i.e. $r = \max\{i | l_i(j-1) \leq L^*\}$. If there is no such machine, we set $r = 0$. Obviously, $r \neq n$, otherwise $n \in S$, since $l_n(j-1) + p_n(j) \leq L^* + L^* \leq 2L^*$ as job j can incur at most L^* load on the fastest machine n. This would contradict $S = \emptyset$. Thus $r < n$.

Define $\Gamma = \{i | i > r\}$, the set of *overloaded* machines in S. Since $r < n$, $\Gamma \neq \emptyset$. Denote by $\mathcal{S}_i$ and by $\mathcal{S}_i^*$ the sets of jobs assigned to machine i by the online and offline algorithms, respectively. Consider the load on all machines in Γ as assigned by the online algorithm:

$$\begin{aligned}
\sum_{i \in \Gamma, s \in \mathcal{S}_i} p_n(s) &= \sum_{i \in \Gamma} [\sum_{s \in \mathcal{S}_i} [\frac{p_n(s)}{p_i(s)} p_i(s)]] \\
&= \sum_{i \in \Gamma} [\sum_{s \in \mathcal{S}_i} [\frac{v_i}{v_n} p_i(s)]] && (2) \\
&= \sum_{i \in \Gamma} [\frac{v_i}{v_n} \sum_{s \in \mathcal{S}_i} [p_i(s)]] && (3) \\
&> \sum_{i \in \Gamma} [\frac{v_i}{v_n} L^*] \\
&\geq \sum_{i \in \Gamma} [\frac{v_i}{v_n} \sum_{s \in \mathcal{S}_i^*} [p_i(s)]] && (4) \\
&= \sum_{i \in \Gamma} [\sum_{s \in \mathcal{S}_i^*} [\frac{v_i}{v_n} p_i(s)]] \\
&= \sum_{i \in \Gamma} [\sum_{s \in \mathcal{S}_i^*} [\frac{p_n(s)}{p_i(s)} p_i(s)]] \\
&= \sum_{i \in \Gamma, s \in \mathcal{S}_i^*} p_n(s) && (5)
\end{aligned}$$

At line 2, note that we are dealing with *related* machines. At line 3, note that $\sum_{s \in \mathcal{S}_i} p_i(s)$ is just the load assigned to machine i by the *online* algorithm, which must be more than L^* since $i \in \Gamma$. At line 4, note that the *offline* algorithm assigned not more than L^* load to any machine in Γ.

The result at line 5 implies that there exists a job $s \in \cup_{i \in \Gamma} \mathcal{S}_i$, such that $s \notin \cup_{i \in \Gamma} \mathcal{S}_i^*$, i.e. there exists a job assigned by the online algorithm to a machine $i \in \Gamma$, and assigned by the offline algorithm to a slower machine $i' \notin \Gamma$.

Because the offline algorithm was able to fit this job s on machine i', $p_{i'}(s) \leq L^*$. Since $r \geq i'$ (i.e. machine r is at least as fast as machine i') then $p_r(s) \leq L^*$. Since job s was assigned before job j (recall that job j is the job we are trying to assign at the current step in the algorithm) then $l_r(s-1) \leq l_r(j-1) \leq L^*$. But this means that the online algorithm would have placed job s on machine r or a slower machine instead of on machine $i \in \Gamma$, which is a contradiction. ▮

If we know the optimal load L^*, the above online algorithm assigns jobs to machines so that the maximum load never exceeds $2L^*$. Since for every job j the set S is always non-empty, there is always a machine to which the job can be assigned without exceeding the maximum load $2L^*$ (i.e the algorithm can never get 'stuck'). Hence the algorithm has a competitive ratio of 2.

Now since we do not know L^*, we start with an estimate of the value and then refine it as we go along. Denote our i^{th} estimate as L_i^*. At any stage, if we underestimate the value of L^*, then we would come to a situation where the set $S = \emptyset$. To start with a low estimate, let L_1^* be the load of the first job on the fastest machine. At each job assignment, if $S = \emptyset$ (we cannot assign this job), we simply update our estimate: let $L_{i+1}^* = 2L_i^*$. This ensures that our final estimate of L^*, say L_k^* is within a factor of 2 of the actual value. The total load is

$$\begin{aligned} 2L_1^* + 2L_2^* + ... + 2L_k^* &\leq 4L_k^* \\ &\leq 8L^* \end{aligned}$$

giving us an approximation factor of 8.

4 Yao's Minimax Theorem from LP duality

Here we prove Yao's minimax principle using LP duality. Recall that we use Yao's minimax principle to lower bound the running time over randomized algorithms by proving a lower bound on the running time of every deterministic algorithm for a particular input distribution.

We first show the following result:

Lemma 4.1 *Suppose $A \in \mathrm{R}^{m\times n}$ is a given matrix. Then:*

$$\min_y \max_x y^T Ax = \max_x \min_y y^T Ax$$

where $x \in \mathrm{R}^n$ and $y \in \mathrm{R}^m$ are probability distributions.

Proof: We will prove this via LP duality. First, however, observe that for fixed y, $\max_x (y^T A)x$, where x is a probability distribution, is the same as $\max_i (A^T y)_i$, since $(y^T A)x$ is just a weighted average of the elements of $(A^T y)$. Similarly, $\min_y y^T Ax = \min_i (Ax)_i$. Therefore, the problem of finding $\min_y \max_x y^T Ax$ over all probability distributions is the same as $\min_y \max_i (A^T y)_i$, which can be written as the following LP:

$$\begin{array}{ll} \min. & t \\ \text{s.t.} & \sum_i y_i A_{ij} \le t, \quad \forall j \\ & \sum_i y_i = 1, \\ & y_i \ge 0. \ \ \forall i \end{array} \tag{6}$$

The dual of this LP is:

$$\begin{array}{ll} \max. & w \\ \text{s.t.} & \sum_j A_{ij} x_j - w \ge 0, \quad \forall i \\ & \sum_j x_j = 1, \\ & x_j \ge 0. \ \ \forall j \end{array} \tag{7}$$

Which can be simplified to:

$$\begin{array}{ll} \max. & w \\ \text{s.t.} & \sum_j A_{ij} x_j \ge w, \quad \forall i \\ & \sum_j x_j = 1, \\ & x_j \ge 0. \ \ \forall j \end{array} \tag{8}$$

This is exactly $\max_x \min_j (Ax)_j$ in LP form, which we have already shown is equivalent to finding $\max_x \min_y y^T Ax$, for y a probability distribution. By strong duality, these two quantities are equal, as desired. ▮

Now suppose $A \in R^{m\times n}$ is a matrix where $A(i,j)$ is the running time of algorithm i on input j, where m is the number of algorithms and n is the number of possible inputs. If x and y are probability distributions on the inputs and algorithms respectively, then $y^T Ax$ is the expected running time with input distribution x and algorithm distribution y (a distribution on algorithms y can be thought of as specifying a randomized algorithm). By Lemma 4.1, we have that:

$$\min_y \max_i (A^T y)_i = \max_x \min_j (Ax)_j,$$

[3]Edited by Joshua Wang.

This implies that for every y and x:

$$\max_i (A^T y)_i \geq \min_j (Ax)_j.$$

Suppose we have a distribution x on inputs, for which the running time for the optimal deterministic algorithm (for this distribution of inputs), $\min_j (Ax)_j$, is at least L (*i.e.*, every deterministic algorithm has expected running time at least L for this distribution). Then L is also a lower bound on the running time of every randomized algorithm, *i.e.* for every randomized algorithm, there exists a specific input such that the running time of the randomized algorithm is at least L. This is exactly Yao's Minimax Principle!

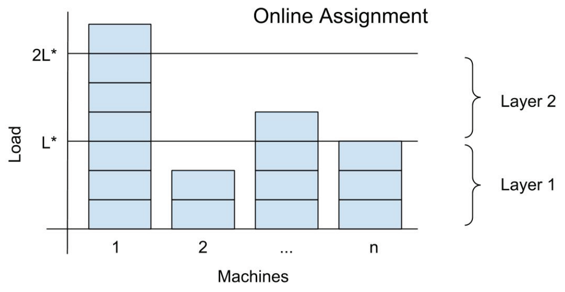
5 One-Infinity Case

We shall now consider the $1/\infty$ model, in which every job has a load vector whose entries are either 1 or ∞. In particular, if for machine i and job j, $p_i(j) = \infty$, then job j *cannot* be scheduled on machine i. We will prove that the greedy algorithm achieves a competitive ratio of $O(\log n)$. Furthermore we will prove a lower bound of $\Omega(\log n)$ on the competitive ratio of any online algorithm for this model. This lower bound also applies to randomized online algorithms. The results presented in this section are mostly taken from [8].

5.1 The Greedy Algorithm

Theorem 5.1 *The greedy algorithm achieves a competitive ratio of* $\lceil \log_2 n \rceil + 1$.

Proof: Let L^* be the maximum load on a machine in an optimal offline solution, denoted by OPT. If the total number of jobs is W then $L^* \geq \frac{W}{n}$. Let M be the assignment of jobs to machines produced by the greedy algorithm. Let $j_1, j_2, \ldots, j_k$ be the jobs assigned to machine i in M, sorted in the order that the greedy algorithm assigned them. We divide jobs into *layers* as follows: The first L^* jobs in the sequence belong to layer 1, the next L^* to layer 2, and so on. Since each layer needs to be completely filled before beginning the next layer, all layers above layer $\lceil \frac{k}{L^*} \rceil$ will have no jobs for machine i. For each machine i, divide the jobs into layers as described above.



Using this definition of layers, we make the following definitions:

- R_i - The set of jobs assigned in layers above i.
- W_i - The set of jobs assigned in layer i.
- W_{ij} - The set of jobs assigned to machine j in layer i.
- O_j - The set of jobs assigned in OPT to machine j.
- O_{ij} - The subset of O_j assigned by the *online* algorithm in layers above i (in other words, $O_j \cap R_i$).

By definition $|R_0| = W$. We prove that the size of R_i decreases exponentially as i increases.

Lemma 5.2 *For every layer i and machine j*, $|W_{ij}| \geq |O_{ij}|$.

[3]Edited by Joshua Wang.

Proof: If $O_{ij} = \emptyset$, the Lemma follows immediately.

The Lemma also follows if $|W_{ij}| = L^*$, since this means layer i is full of assigned jobs for machine j, and since OPT assigned at most L^* jobs to any machine, then $|O_{ij}| \le |O_j| \le L^*$.

Therefore, the only remaining case is where $|W_{ij}| < L^*$ and $O_{ij} \neq \emptyset$. We claim that this case cannot occur. $|W_{ij}| < L^*$ means that in the i^{th} layer, machine j is not completely loaded, implying that in all layers above i the load on machine j is 0. Hence the total load on machine j must be strictly less than $i \cdot L^*$. Since $O_{ij} \neq \emptyset$, the jobs in O_{ij} must have been assigned in M to machines other than j. Let r be such a machine, i.e. there is a job $k \in O_{ij}$ assigned to machine r, with $r \neq j$. Since job k appears in a layer above i (by definition of O_{ij}), then the load on machine r at the time of assignment must have been at least $i \cdot L^*$. However, the load on machine j at the time of assignment could be at most the total load on machine j, which is strictly less than $i \cdot L^*$. Since the assignment M was produced by the greedy algorithm, the load of machine r could not have been greater than the load of machine j at the time of the assignment. This gives a contradiction. ▮

Lemma 5.3 *$\forall i$: $|R_i| \le \frac{1}{2}|R_{i-1}|$*

Proof: Note that $R_i = \cup_{j=1}^{n} O_{ij}$. Also, for fixed i and $1 \le j \le n$, the sets O_{ij} are mutually disjoint. Thus, by Lemma 5.2,

$$|W_i| = \sum_{j=1}^{n} |W_{ij}| \ge \sum_{j=1}^{n} |O_{ij}| = |R_i|$$

This implies that $|R_i| = |R_{i-1}| - |W_i| \le |R_{i-1}| - |R_i|$. Hence, $|R_i| \le \frac{1}{2}|R_{i-1}|$. ▮

We are now ready to complete the proof of the Theorem. Let $b = \lceil \log_2 n \rceil$. Applying Lemma 5.3 b times, we obtain that $|R_b| \le \frac{1}{n}|R_0| = \frac{W}{n} \le L^*$. Hence, the load on a machine can be at most $L^* \cdot b + |R_b| \le L^* \cdot (b+1)$. Since OPT achieves a load of L^*, we have proved that the competitive ratio of greedy is at most $\lceil \log_2 n \rceil + 1$. ▮

5.2 Lower bound

Theorem 5.4 *The competitive ratio of any online load-balancing algorithm for the $1/\infty$ model is at least $\frac{1}{2}\lceil \log_2(n+1) \rceil$.*

Proof: For simplicity and without loss of generality assume that n is a power of 2, i.e. $n = 2^{k-1}$. For a job j, let $M_j = \{i | p_i(j) = 1\}$. Thus M_j is the set of machines on which job j can be scheduled. A sequence σ of jobs with the following properties is constructed by an oblivious adversary based on the decisions that will be made by the deterministic online load-balancing algorithm:

1. The sequence σ is logically divided into k concatenated sub-sequences, $\sigma = \sigma_1 \cdot \sigma_2 \cdot \ldots \cdot \sigma_k$. Each sub-sequence can be thought of as a step. In step i, $1 \le i \le k$, the sub-sequence σ_i is presented to the online algorithm. For $1 \le i \le k-1$, $|\sigma_i| = \frac{n}{2^i}$ and define $|\sigma_k| = 1$. Note that this is consistent with the total number of jobs we have, since it follows that $|\sigma| = \sum_{i=1}^{k} |\sigma_i| = n\left(\frac{1}{2} + \frac{1}{4} + \cdots + \frac{1}{2^i} + \cdots + \frac{1}{2^{k-1}}\right) + 1 = (2^{k-2} + 2^{k-3} + \cdots + 1) + 1 = 2^{k-1} = n$.

[3]Edited by David Jia.

2. For each job $j \in \sigma_i$ we have $M_j = Y_i$, where Y_i is a subset of machines chosen by the adversary for step i. (Note that, in order for the adversary to be able to restrict those machines to which the online algorithm can assign jobs, the adversary must be able to define the load vector $\vec{p}(j)$ for each job $j \in \sigma$.)

3. $Y_{i+1} \subset Y_i$ and $|Y_i| = \frac{n}{2^{i-1}}$. The intuition here is that at each subsequent step, we limit the machines that can perform the jobs in that step to be a subset of the machines in the previous step, more precisely, to half of the machines in the "eligible" machines (machines $i \in M_i$) of the previous step.

4. At the beginning of step i, before the jobs in σ_i are presented, the average load of the machines in Y_i is at least $\frac{i-1}{2}$. In other words, in the next step after step $i-1$, we choose "eligible" machines to be only from the set of machines to which the online algorithm has assigned jobs in the previous step. This process of choosing Y_i will be explained further later on.

If a sequence with the above properties can be constructed, then at the end of step $k-1$, there is a machine in Y_k whose load is at least $\frac{k-1}{2}$, because of property 4 above. Since $|Y_k| = 1$ and $|\sigma_k| = 1$, there is only one machine that can handle the last job. The load of that machine will be increased to $\frac{k-1}{2} + 1 = \frac{k+1}{2}$. On the other hand, if we remove the requirement in property 4 above, there is an assignment of jobs to machines such that the maximum load is 1, since all jobs in σ_i can be assigned to the machines (on which σ *does not* recurse) in $Y_i - Y_{i+1}$ (define $Y_{k+1} = \emptyset$). To see this, note that $Y_{i+1} \subset Y_i$ and $|Y_i| - |Y_{i+1}| = \frac{n}{2^{i-1}} - \frac{n}{2^i} = \frac{n}{2^i} = |\sigma_i|$. So all jobs $j \in \sigma_i$ can be assigned to a machine in Y_i that becomes an ineligible or ∞ machine for jobs in the next step. Hence, the competitive ratio of the online algorithm is at least $\frac{k+1}{2} \geq \frac{1}{2}\lceil \log_2(n+1) \rceil$.

Now, it is enough to show the construction of sequence sigma σ . To construct σ, we need to describe how set Y_i is chosen so that the above properties are maintained. The set Y_1 is defined to be M, the set of all machines. If $S = \{n'+1, n'+2, \ldots, n'+2l\}$ is a set of $2l$ machines, let $low(S)$ be the set of machines in S with the smallest l numbers, and $high(S)$ be the set of machines with the highest l numbers. $S = low(S) \cup high(S)$ and $low(S)$ is disjoint from $high(S)$. Thus $low(S) = \{n'+1, n'+2, \ldots, n'+l\}$ and $high(S) = \{n'+l+1, n'+l+2, \ldots, n'+2l\}$. To obtain Y_{i+1}, choose from the two sets $low(Y_i)$ and $high(Y_i)$, that set which will have a higher average load and let Y_{i+1} be this set. Properties 1 and 2 can always be maintained independently of Y_i, and property 3 clearly holds. We prove property 4 by induction on the steps of the algorithm. It obviously holds for $i = 1$. Assume inductively, that it holds for i, i.e. at the beginning of step i the average load on a machine in Y_i is at least $\frac{i-1}{2}$. Hence the sum of the loads on the machines in Y_i is at least $\frac{i-1}{2}|Y_i|$. In step i, all of the jobs of σ_i must be assigned to machines in Y_i, so that at the end of step i, the sum of the loads of the machines is at least $\frac{i-1}{2}|Y_i| + |\sigma_i| = \frac{i-1}{2}|Y_i| + \frac{1}{2}|Y_i| = \frac{i}{2}|Y_i|$. Thus the average load per machine in Y_i is $\frac{i}{2}$. Hence the average load on one of the halves $low(Y_i)$ or $high(Y_i)$ must be at least $\frac{i}{2}$. Thus the average load on Y_{i+1} must be at least $\frac{i}{2}$. ▮

To give a better intuition, one can think about the adversary as one that is trying to slowly force the online algorithm into loading a machine that currently has maximum load. The adversary does this by constantly choosing the worst possible job allocation, even when the online algorithm does it best to alleviate the problems (i.e. distribute the jobs evenly amongst all "eligible" machines) that is, by picking the "eligible" machines in each step only from the ones that already have considerable load.

In the following example, a sequence of steps is demonstrated with $k = 4$ and $n = 2^{k-1} = 8$.

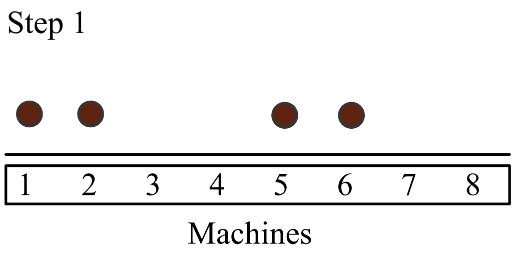


Figure 7: Here in Step 1, $Y_1 = \{1, 2, \dots, 8\}$, i.e. the whole set of machines. The online algorithm chooses to allocate the $\frac{n}{2} = 4$ jobs to machines 1, 2, 5, and 6. The average load of Y_1 is $\frac{1-1}{2} = 0$.

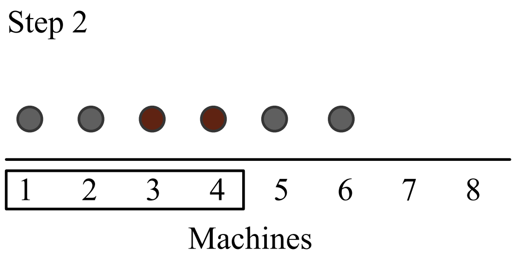


Figure 8: In Step 2, the side that has the highest average load is chosen. In this case both sides are the same, so we choose arbitrarily. $Y_2 = \{1, 2, 3, 4\}$. The online algorithm chooses to allocate the $\frac{n}{2^2} = 2$ jobs to machines 3, and 4. The average load of Y_2 is at least $\frac{2-1}{2} = \frac{1}{2}$.

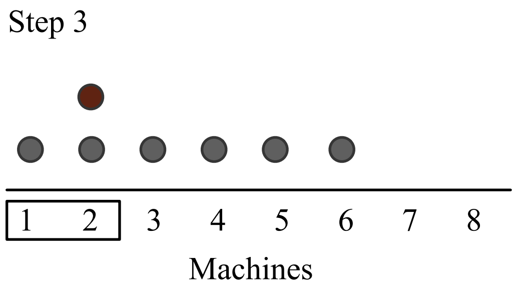


Figure 9: In Step 3, the side that has greater average load is again chosen. $Y_3 = \{1, 2\}$. The online algorithm chooses to allocate the $\frac{n}{2^3} = 1$ jobs to machine 3. The average load of Y_3 is at least $\frac{3-1}{2} = 1$.

Randomized lower bound

The above proof can also be adapted to prove a lower bound of $\Omega(\log n)$ on the competitive ratio of a randomized algorithm for online load-balancing in the $1/\infty$ model (against an oblivious adversary). We can produce a distribution on inputs such that every deterministic algorithm gives an assignment of jobs to machines with *expected* maximum load $\frac{1}{2}\lceil \log_2(n+1) \rceil$. Also, for every input in this distribution, the offline optimal algorithm can produce an assignment of jobs to machines such that every machine has load at most 1. By Yao's Minimax Principle, this implies the claimed randomized lower bound for oblivious adversaries.

The construction that we use in this part, is similar to what we had in the proof of lower bound for deterministic online algorithms. However, now we cannot use the properties of any specific algorithm to

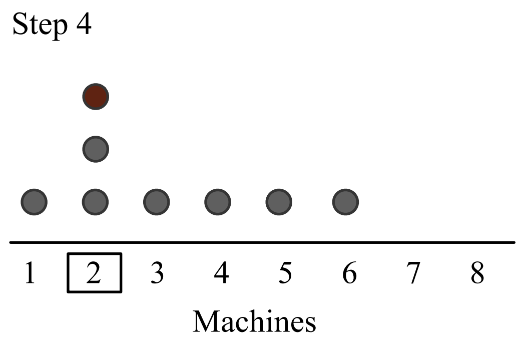


Figure 10: Finally, in Step 4, $Y_3 = \{2\}$ and one last job is allocated to machine 2. The average load of Y_3 is at least $\frac{k+1}{2} = \frac{5}{2}$.

generate the sequence σ. Recall that in the above proof, we selected Y_{i+1} to either of $low(Y_i)$ or $high(Y_i)$ depending on the choices made by the online algorithm. Instead of this, we set Y_{i+1} to either of $low(Y_i)$ or $high(Y_i)$, with equal probability. Note that the sequence σ is completely determined once the sets Y_i are determined. Thus this gives a probability distribution on the set of job sequences σ. Also, we maintain the following modified version of property 4 in the proof: At the beginning of step i, before the jobs in σ_i are presented, the *expected* load on each machine in Y_i is at least $\frac{i-1}{2}$. The original proof goes through almost unchanged. However, in proving that the modified property 4 is maintained, our argument uses expected loads instead of average loads.

Assume inductively, that it holds for i, i.e. at the beginning of step i the expected load on a machine in Y_i is at least $\frac{i-1}{2}$. Hence the expected sum of loads on the machines in Y_i is at least $\frac{i-1}{2}|Y_i|$. In step i, all of the jobs of σ_i must be assigned to machines in Y_i, so that at the end of step i, the expected sum of the loads of the machines is at least $\frac{i-1}{2}|Y_i| + |\sigma_i| = \frac{i-1}{2}|Y_i| + \frac{1}{2}|Y_i| = \frac{i}{2}|Y_i|$. Let E_{low} and E_{high} be the expected loads per machine in $low(Y_i)$ and $high(Y_i)$ respectively. Then the expected sum of the loads on machines in Y_i is given by

$$E_{low}\frac{|Y_i|}{2} + E_{high}\frac{|Y_i|}{2} \geq \frac{i}{2}|Y_i|$$

Hence $\frac{1}{2}E_{low} + \frac{1}{2}E_{high} \geq \frac{i}{2}$. Now Y_{i+1} is chosen to be either of $low(Y_i)$ or $high(Y_i)$ with equal probability. Then, at the beginning of step i, the expected load on machines in Y_{i+1} is $\frac{1}{2}E_{low} + \frac{1}{2}E_{high}$ which is $\geq \frac{i}{2}$.

Now as before, at the end of step k, the expected load on Y_k is at least $\frac{k+1}{2}$. Thus the expected value of the maximum load is $\frac{k+1}{2}$. However, the offline optimal algorithm simply assigns the jobs in σ_i to the machines in $Y_i - Y_{i+1}$, maintaining a maximum load of 1.

6 On-line scheduling on unrelated machines

In the last lecture, we studied on-line scheduling on unrelated machines in the special case where for each job, the entries of the load vector are either 1 or ∞ i.e. for each job some machines are eligible and can perform the job with one unit of load and some machines are ineligible and can not perform the job. Here, a greedy algorithm is $O(\log n)$-competitive [8]. In the case where the load vector consists of arbitrary values, the greedy algorithm fails to achieve better than a $O(n)$ competitive ratio [3]. We now present a different algorithm that achieves a competitive ratio of $O(\log n)$. This result was derived in [3], but the proof technique we use is from [6].

Let $l_i(j)$ denote the load on machine i after j jobs have been submitted to the online algorithm. Let $p_i(j)$ be the load of job j on machine i. Define $\Delta_i(j)$ as

$$\Delta_i(j) = a^{l_i(j-1)+p_i(j)} - a^{l_i(j-1)},$$

where a is a constant to be defined later. We will show that the following algorithm achieves a competitive ratio of $O(\log n)$:

Algorithm A: Assign job j to the machine i that minimizes $\Delta_i(j)$.

Intuition: Unlike the greedy algorithm, which tries to minimize $l_i(j-1) + p_i(j)$, algorithm **A** minimizes, in some sense, the *difference* between the previous load and the new load, where *difference* is defined, as we will see, in some non-linear way. The reason the greedy algorithm that utilizes a linear definition of *difference* performs poorly is because it tries to equalize the load on all machines, and so an adversary can force it to make the wrong choice of machine each time [3]. Observe that if there is a job j such that it has the same load on each machine, that is, $p_i(j) = p(j)$ for all i (for this job j), then **A** makes the same choice as a greedy algorithm: it chooses the i minimizing $a^{l_i(j-1)}(a^{p(j)} - 1)$, which is the least loaded machine. When a job does not place the same load on all machines, the exponentiation might lead to a non-greedy choice. For example, consider a situation where $p_i(j) \ll p_{i'}(j)$. In this case, the load on i has to be much larger than the load on i' in order for the algorithm **A** to put job j on i'.

Lemma 6.1 *If job j is assigned to machine i by the online algorithm, and k is the number of jobs, then for any machine q,*

$$\Delta_i(j) \le a^{l_q(k)+p_q(j)} - a^{l_q(k)}$$

Proof: Since job j is assigned to machine i, $\Delta_i(j) \le \Delta_q(j)$ for any machine q, and so

$$\begin{aligned} \Delta_i(j) &\le a^{l_q(j-1)}(a^{p_q(j)} - 1) \\ &\le a^{l_q(k)}(a^{p_q(j)} - 1) \\ &= a^{l_q(k)+p_q(j)} - a^{l_q(k)}, \end{aligned}$$

since $l_q(k) \ge l_q(j-1)$ for any j. Note that this lemma relies on the convexity of a^x; this lemma could not have been obtained if a non-convex function were used to define Δ. ▮

Theorem 6.2 *The online algorithm **A** is $O(\log n)$-competitive.*

Proof: Let $i^*(j)$ be the machine that the optimal, offline algorithm assigns job j to. From Lemma 6.1, we have

$$\Delta_i(j) \le a^{l_{i^*(j)}(k)+p_{i^*(j)}(j)} - a^{l_{i^*(j)}(k)}, \tag{9}$$

[3]Edited by David Jia.

where i is the machine to which job j is assigned by the online algorithm **A**.

Summing over all j, yields

$$\sum_j a^{l_i(j-1)}(a^{p_i(j)}-1) \le \sum_j a^{l_{i^*(j)}(k)}(a^{p_{i^*(j)}(j)}-1) \tag{10}$$

Let $a = 1+\gamma$. We have

$$a^x - 1 \le \gamma x, \qquad \forall x \in [0,1]. \tag{11}$$

which can be derived from taking the first order Taylor series expansion of $a^x - 1$. Now assume that $OPT = 1$. Effectively, this means that we scale all values of $p_i(j)$ by OPT by normalizing the load vector of all possible jobs. Later on, we will show how to do a binary search on the value of OPT which costs us only a constant factor in the competitive ratio.

Since $OPT = 1$, each $p_i(j) \le 1$. Therefore the inequality in (11) holds for $x = p_i(j) \in [0,1]$, and we can replace the R.H.S of equation (10) by $\gamma \sum_j a^{l_{i^*(j)}(k)} p_{i^*(j)}(j)$.

Now take a sum over machines instead of jobs, we get

$$\begin{aligned} R.H.S &= \gamma \sum_i \sum_{j:i^*(j)=i} a^{l_{i^*(j)}(k)} p_{i^*(j)}(j) \\ &\le \gamma \sum_i a^{l_i(k)} \end{aligned} \tag{12}$$

The last inequality holds because the inner sum is merely the total load in the offline algorithm on machine i, which is $\le OPT = 1$, and we know that $0 \le p_i(j) \le 1$ and the sum of the loads, of the online algorithm is by definition greater than or equal to that of the OPT. Now,

$$\begin{aligned} L.H.S &= \sum_j a^{l_i(j-1)}(a^{p_i(j)}-1) \\ &= \sum_i \sum_{j:j \ assigned \ to \ i} a^{l_i(j-1)}(a^{p_i(j)}-1) \\ &\overset{(a)}{=} \sum_i (a^{l_i(k)}-1) \\ &= \sum_i a^{l_i(k)} - n, \end{aligned} \tag{13}$$

where (a) is because $l_i(j-1) + p_i(j) = l_i(j) = l_i(j'-1)$, where j' is the next job after job j to be assigned to machine i (so that we have a telescoping sum).

Combining equations (12) and (13) yields

$$\sum_i a^{l_i(k)} - n \le \gamma \sum_i a^{l_i(k)} \implies \sum_i a^{l_i(k)} \cdot (1-\gamma) \le n$$

which gives

$$\sum_i a^{l_i(k)} \le \frac{n}{1-\gamma}.$$

Therefore, each $a^{l_i(k)} \leq \frac{n}{1-\gamma}$, and specifically,

$$\max l_i(k) \leq \log_a \frac{n}{1-\gamma} \tag{14}$$

Choosing $\gamma = 0.5$ (and $a = 1 + \gamma = 1.5$) yields a competitive ratio of $\log_{1.5} 2n$.

Notice here that what we're really saying is

$$\max l_i(k) \leq \log_a \frac{n}{1-\gamma} OPT \tag{15}$$

but earlier we have assumed that $OPT = 1$, which yields (14). We will not relax this assumption by searching for a suitable OPT, regardless of which, since we are normalizing all load vectors by this amount, will not change our algorithm.

Determining the value of OPT. Pick an initial value of $OPT = \lambda$. Run the algorithm scaling all values by λ. If the maximum value of $l_i(j)$ crosses $\lambda \log n$, we clearly have chosen a wrong value for OPT because this would mean that our competitive ratio given in equation (15) above is inconsistent. Set $\lambda_{new} = 2\lambda$, and repeat the algorithm, *assuming that the new initial loads on the machines are all 0.* Do this until we obtain a solution $S \leq \lambda \log n$.

How much have we lost by this procedure ? Each time we double λ, we could have been just above $\lambda \log n$, so we introduce an error $\epsilon <= \lambda$. We lose a factor of 2 over OPT by summing up the errors from all doublings, and we lose an extra factor of 2 in the last stage because of our imprecise approximation. Therefore, we lose a factor of 4 overall.

What remains is to choose the starting value of λ. We choose $\lambda = \min_i p_i(1)$. ▮

The main intuition in this algorithm is that we want to distinguish the cases when a particular job slightly increases load on one already heavily loaded machine versus significantly increasing load on another not very loaded machine. This way, can save the less loaded machine for later jobs that can better utilize it by requiring less load on that machine. More formally, in the unrelated machines case, given a job j, we sometime want to choose a machine i over a machine i' even though $\ell_i(j-1)+p_i(j) \geq \ell_{i'}(j-1)+p_{i'}(j)$ if $\ell_i(j-1) \gg \ell_{i'}(j-1)$ but $p_i(j) \ll p_{i'}(j)$. We can do this by mapping their difference to an exponential (or any convex) function. For the same difference of a pair of values, the difference in the value of the convex function, $\Delta_i(j)$ will allow us to distinguish which one is "better" by picking the machine that has the better comparative advantage for that particular job.

To be more precise, note that the goal of the problem is to minimize $\max_q \ell_q(k)$, where k is the total number of jobs. This is not a smooth function, so we will try to replace it with a nicer function with similar properties. Consider $\Phi = \sum_q a^{\ell_q(k)}$, and suppose that we were able to minimize Φ. Then if L^* is the optimal load, we certainly have $a^{L^*} \leq \Phi_{\min}$, as the most loaded machine has load at least L^*, and $\Phi_{\min} \leq na^{L^*}$, since in the offline algorithm all n machines do not exceed load L^*.

Suppose we have some distribution of loads which attains $\Phi_{\min}$. Then if L is the maximum load in this allocation, we certainly have $a^L \leq \Phi_{\min} \leq na^{L^*}$ and therefore $L \leq \log_a n + L^*$. Hence, the load cannot be much worse than L^*, so it is reasonable to be minimizing Φ instead of $\max_q \ell_q(k)$. However, when we allocate job j to machine q, we increase Φ by precisely the quantity $a^{\ell_q(j-1)+p_q(j)} - a^{\ell_q(j-1)}$, so our algorithm is always choosing the machine which results in the smallest increase to Φ.

In the example illustrated on the right, we see that a machine 1 currently has a load of 10 while machine 2 has a load of 1. By adding the new job j, machine 1's load will increase by 1 while machine 2's load will increase by 9. In the original linear *difference*, we would have chosen to put job j on machine 2. But this would have clogged up machine 2 while we could have just increased machine 1's load slightly. In our new algorithm as presented in this section, we would choose machine 1 as clearly $1.5^{10+1} - 1.5^{10} \approx 28.83 < 56.16 \approx 1.5^{10} - 1.5^{1}$.

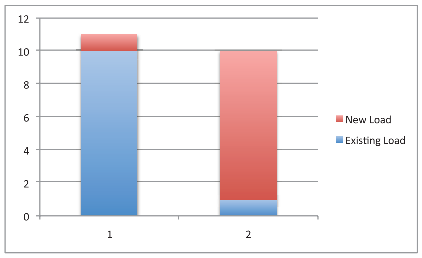


Figure 11: Here we choose machine 1 even though its new load exceeds that of machine 2.

7 Application of this algorithm to on-line network routing

Consider a network where each edge has an associated capacity (bandwidth). A request consists of three entries: a source node, a destination node, and the bandwidth required for the transmission. The network then sets up a connection by choosing a path between the source and the destination and reserving bandwidth along it [3]. Similarly to the load balancing setting were we were not allowed to move a job once it is allocated to a machine, in the routing model we are not allowed to change the path used to satisfy a request.

There are two natural things we may wish to optimize. The first is to require that no capacity constraints are violated and attempt to maximize the fraction of satisfied requests. The second is to require that all requests are satisfied, and minimize the maximum ratio by which our allocation exceeds any edge capacity. This latter model, called the *congestion minimization model*, is the one we will consider here.

Formally, we have a graph $G = (V, E), |V| = n, |E| = m$, and a capacity function $u : E \to \mathbb{R}^+$. Incoming requests are of the form $R_i = (u_i, v_i, p(i))$, where $u_i, v_i \in V$ are the source and destination nodes, and $p(i) \in \mathbb{R}^+$ is the desired capacity of the link. We normalize bandwidth requests w.r.t edge capacities, and therefore define

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

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be the set of paths assigned by the online algorithm for requests 1 through k, and let $\mathcal{P}^* = \{P_1^*, P_2^*, \ldots, P_k^*\}$ be the paths assigned by the optimal offline algorithm. Now, given a set of paths $\mathcal{P}$, we define the *relative load* after the first j requests by

$$\ell_e(j) = \sum_{\substack{i \le j \\ e \in P_i}} p_e(i).$$

Let $\lambda(j) = \max_{e \in E} \ell_e(j), \lambda = \lambda(k)$. Similarly define $\lambda^*(j), \lambda^*, \ell_e^*(j)$ for the corresponding offline quantities. The problem is now to minimize λ/λ^*.

Note: As before, we will assume that $\lambda^* = 1$, and use a "doubling" argument to determine the correct value.

We will proceed as in the unrelated machines algorithm, by minimizing the objective $\Phi = \sum_{e \in E} a^{\ell_e(i)}$. In other words, given a path P_i, we define

$$W_i = \sum_{e \in P_i} \left(a^{\ell_e(i-1) + p_e(i)} - a^{\ell_e(i-1)} \right)$$

Algorithm B: Given a request i, choose a path minimizing W_i. Note that this path can be determined easily using a shortest-path algorithm with the expression inside the summation as the cost of an edge.

Theorem 7.1 *This algorithm is $O(\log n)$-competitive. [3]*

[3]Edited by Jeffrey Chen.

Proof: As in Lemma 6.1, for every path P' from s_i to t_i, we have

$$W_i \le \sum_{e \in P'} \left(a^{\ell_e(k)+p_e(i)} - a^{\ell_e(k)} \right)$$

In particular, this holds for P_i^*, the i-th path chosen in the optimal algorithm. As before, we set $a = 1+\gamma$, and since $\lambda^* = 1$, $p_e(i) \le 1$ for all e. Then we have

$$\begin{aligned} W_i &\le \sum_{e \in P_i^*} a^{\ell_e(k)} \left(a^{p_e(i)} - 1 \right) \\ &\le \sum_{e \in P_i^*} a^{\ell_e(k)} \left(\gamma p_e(i) \right) \\ &= \gamma \sum_{e \in P_i^*} a^{\ell_e(k)} p_e(i) \end{aligned}$$

Summing over i, we then have

$$\begin{aligned} \sum_i W_i &\le \gamma \sum_i \sum_{e \in P_i^*} a^{\ell_e(k)} p_e(i) \\ &= \gamma \sum_e a^{\ell_e(k)} \ell_e^*(k) \end{aligned}$$

But $\ell_e^*(k) \le \lambda^* \le 1$, so we conclude that

$$\begin{aligned} \sum_i W_i &\le \gamma \sum_e a^{\ell_e(k)} \\ &= \gamma \Phi \end{aligned}$$

Now recall that W_i is precisely the change in Φ when the online algorithm processes the i-th request, and the value of Φ before any requests have been processed is $\sum_e a^0 = m$, so $\sum_i W_i$ is simply $\Phi - m$. Thus, we conclude that $\Phi - m \le \gamma\Phi$, so $\Phi \le \frac{m}{1-\gamma}$. In particular, this implies that $L \le \log_a \frac{m}{1-\gamma} = O(\log m)$.

▮

Discussion: No lower bound is known for this problem. In the directed graph version, there is a known lower bound of $\Omega(\log n)$ [8].

8 Online Load Balancing of Temporary Tasks

In this section we consider Online Balancing problem - the jobs are now *temporary*. The jobs arrive one by one, each having an associated *load vector*, $\vec{p}(j) = (p_1(j), p_2(j), \dots p_n(j))$, where $p_i(j) \geq 0$ and a duration, $d(j)$. Assigning job j to machine i increases the load on this machine by $p_i(j)$ for a period $d(j)$. As before, the load balancing is non-preemptive i.e. no reassignment is allowed.

Since the arriving tasks have to be assigned without the knowledge of future tasks, it is natural to evaluate performance in terms of the *competitive ratio*. In this case, the competitive ratio is the supremum over all input sequences, of the maximum load (because of the limited duration of the jobs the maximum has to be taken both over time and the machines) achieved by the on-line algorithm to that achieved by the optimal off-line algorithm.

As in the case of jobs that were permanent, this problem can be categorized into three classes according to the properties of the load vector.

Identical machines All the coordinates of a load vector are the same.

$$\forall i, i', j : p_i(j) = p_{i'}(j)$$

Related machines The ith coordinate of each load vector is equal to w_j/s_i where the *weight* w_j depends only on the job j and the *speed* s_i depends only on the machine i.

$$\forall i, i', j, j' : p_i(j)/p_{i'}(j) = p_i(j')/p_{i'}(j') = s_{i'}/s_i$$

where s_i is the speed of the ith machine. In other words, each job j has a weight w_j, and that $\forall i, p_i(j) = w_j/s_i$

Unrelated machines This includes all other cases.

8.1 Identical Machines

This was assigned as a homework problem. It is easily shown that Graham's greedy algorithm achieves a competitive ratio of $2 - \frac{1}{n}$ even when the jobs are temporary.

8.2 Related Machines

Upper Bound For the identical machines case, the greedy approach achieved $O(1)$ competitive ratio both in the case when the jobs were permanent as well as when the jobs had limited durations. It is quite natural to ask if the algorithm that achieved constant competitive ratio for related machines when the jobs were permanent could similarly be extended to the current problem.

This conjecture is true. We get a constant competitive ratio albeit a little poorer one. Even though the algorithm is identical to the case with permanent jobs, it is of considerable interest because of the substantially new proof technique required to prove it. The old proof technique breaks down because the jobs are not permanent. The method of getting a contradiction now has to span both time and machines. The essential idea is to isolate groups of machines at different time instants with larger and larger loads until one reaches a contradiction.

[3]Edited by Jack Chen.

It is assumed that the machines are ordered in the ascending order of speeds i.e. machine 1 is the slowest and machine n is the fastest. If G is a group of machines the total power of the group, $S(G) \equiv \sum_{k \in G} s_k$. If J is a set of jobs then total weight, $W(J) \equiv \sum_{j \in T} w_j$.

It shall be assumed that the load generated by the optimal off-line algorithm, OPT is known to the online algorithm. This is not a very binding assumption since one can always use a doubling strategy at the cost of increasing the competitive ration by a factor of 4. Let c be an integer constant (It is shown later that the optimal choice of c is 5). Assume that a task j arrives at time t and the load on machine i just before the arrival of the job is $l_i(t^-)$. A job is said to be *assignable* to machine g iff $w_j/s_g \leq OPT$ and $l_g(t^-) + w_j/s_g \leq c \cdot OPT$

Algorithm 1 : Assign the new job to the*slowest assignable* machine. Break ties by assigning to the machine with the smallest index.

Theorem 8.1 *Provided $c > 5$, Algorithm 1 guarantees that every job is assignable, which implies that the algorithm is c-competitive.*

Proof: Suppose that at some time instant t_0, a job q arrives that is not *assignable*. It shall be shown that this entails a contradiction to the supposed value of OPT by showing that at some time in the past the aggregate weight of the tasks in the system was more than $\sum_{i=1}^{n} s_i \cdot OPT$. Starting from the current time t_0, a sequence of times $t_0 > t_1 > \ldots > t_k$ is constructed. At time t_i^- let $J(G_i)$ be the collection of jobs that are active and are assigned to the set of machines G_i by the on-line algorithm.

Initially $G_0 = n$. It is proved by induction on i that the following invariants hold:

1. The G_j's ,$0 < j < k$ are disjoint sets of consecutive machines, i.e.,

$$G_j = m_j, m_{j+1}, \ldots, m_{j-1} - 1$$

for some m_j's. Some of the G_j's could be empty.

2. At time t_i^- each machine in G_i has a load that exceeds $(c-1)OPT$.

3. For $0 < j \leq k$, $S(G_i) \geq (c-3)S(G_{i-1})$.

Assume that the invariants stated above hold and that the construction (described in the proof below) has been carried out until machine $1 \in G_k$. We know that the cumulative weight of tasks in $J(G_k)$, $W(J(G_k)) > (c-1)S(G_k)OPT$. Now $\sum_{i=1}^{k} S(G_i) \leq 2S(G_k)$. The contradiction follows from the fact that at any instant in time the maximum load that can be handled by all the machines is:

$$\sum_{i=1}^{k} S(g_i) \cdot OPT \leq 2S(G_k)OPT < (c-1)S(G_k)OPT$$

since $c > 5$ ▮

Lemma 8.2 *The invariants, stated in the proof above, hold.*

Proof: Set $G_0 = \{n\}$. Since job q cannot be assigned to the fastest machine n and OPT $\geq w_q/s_n$, therefore $l_n(t_0^-) > (c-1)OPT$, which implies that the cumulative weights of the jobs still active on machine n is greater than $(c-1)s_n OPT = (c-1)S(G_0)OPT$. Thus invariant 2 initially holds. The remaining invariants hold trivially.

Now assume that the invariants hold at the end of the ith iteration. The following construction ensures the existence of t_{i+1} and G_{i+1} such that the invariants hold for $i+1$ as well. Since the load on each machine in G_i is greater than $(c-1)OPT$, therefore $W(J(G_i)) > (c-1)S(G_i)OPT$.

Consider machines to which the optimal off-line algorithm assigns the jobs in $J(G_i)$ and let m_{i+1} be the slowest of them. Let

$$G_{i+1} \equiv m_{i+1}, m_{i+1}+1, \ldots, m_i - 1$$

Since all tasks in $J(G_i)$ are active at time t_i^- and the load on any machine never exceeds OPT in the off-line assignment, the cumulative weight of the jobs in $J(G_i)$ assigned by the off-line algorithm to machines in $\bigcup_{0\le j\le i} G_j$ is bounded by $\sum_{0\le j\le i} S(G_j)OPT$, which in turn, by invariant 3, is bounded by $2S(G_i)OPT$. Therefore it must be that, $S(G_{i+1})OPT \ge (c-3)S(G_i)OPT$. As a consequence we get,

$$S(G_{i+1}) \ge (c-3) \cdot S(G_i)$$

Now consider the job q_i with the smallest weight in $J(G_i)$. Define t_{i+1} be the time at which q_i arrived. Since q_i was not assigned to any machine in G_{i+1}, all these machines were unassignable at t_{i+1}. Since $w(q_i)/s_g \le$OPT $\forall g \in G_{i+1}$, therefore

$$\forall g \in G_{i+1} : l_g(t_{i+1}^-) \ge (c-1)OPT$$

This proves that the invariants hold for $i+1$. ▮

Thus, a competitive ratio of 5 is achievable. A proof of $3 - o(1)$ lower bound on the competitive ratio c is given in [7].

8.3 $1/\infty$ scheduling of temporary tasks

In this section, we consider the $1/\infty$ model (identical speed machines with assignment restriction) with temporary jobs of unknown duration. We describe a greedy algorithm with competitve ratio $2\sqrt{n}+1$. We also prove an $\Omega(\sqrt{n})$ lower bound on the competitive ratio. Although this lower bound is the same as that of [9], it is of some interest as it uses a job sequence with the ratio of maximum to minimum ratio that is logarithmic in the number of machines, which implies that a $O(\lg(nT))$ lower bound, where T is the ratio of maximum to minimum job duration, is impossible.

Upper Bound We will now describe an algorithm that achieves an $O(\sqrt{n})$ competitive ratio. At all times, the algorithm maintains an estimate $L(t)$ of OPT satisfying $L(t) \le$ OPT. Let $l_g(t)$ be the load on machine g at time t. A machine is said to be *rich* if $l_g(t) \ge \sqrt{n}L(t)$, and otherwise it is *poor*. In other words, a machine is called rich if it is highly overloaded. Since the jobs are temporary, machines can alternate between being rich and poor. A machine can also become poor since $L(t)$ can increase over the run time of the algorithm. However, since $L(t)$ is constructed to be a monotonically increasing function, a machine can only get poor due to such a change or due to finishing of a job on that machine.

If a machine is rich at t, define its *windfall time* as the last moment in time when it became rich.

Algorithm 2 : Assign the first job to any machine, and set $L(1)$ as the load generated by the first task.

When new task j arrives at time t, set:

$$L(t) = \max \left\{ L(t^-), p_j, \frac{p_j + \sum_g l_g(t^-)}{n} \right\}$$

If possible, assign the new job to a poor machine. Otherwise, if there are no poor machines, assign it to the rich machine with the most recent windfall time.

This algorithm may seem counter-intuitive, because if we are assigning to a rich machine, then we keep allocating jobs to the same machine (the one that became rich last), so we might think that its load would just keep increasing, yielding a very bad competitive ratio. We need to prove that this will not happen. Instead, it will turn out that if the load on the most recently rich machine keeps on growing, then L must eventually increase, which means that other machines may become poor and thus assignable.

Lemma 8.3 *At all times t, the algorithm guarantees $L(t) \leq OPT$ and the number of rich machines at any time is at most $\sqrt{n}$.*

Proof: The proof is by induction on the number of assigned tasks. Initially, $L(1) \leq$OPT. It suffices to consider only the cases when $L(t)$ increases. The claim immediately follows from the fact that $p_j \leq OPT$ and $(p_j + \sum_g l_g(t^-))/n \leq OPT$. Both of these inequalities follow from the fact that the load on any machine in the off-line assignment never exceeds OPT.

Since $nL(t)$ is an upper bound on the aggregate load, at most $\sqrt{n}$ machines can be rich at any point in time. ▮

Theorem 8.4 *The competitive ratio of Algorithm 2 is at most $(2\sqrt{n} + 1)$.*

Proof: Since $L(t) \leq$OPT it suffices to show that at any point t, $l_g(t) \leq \sqrt{n} \cdot (L(t) + OPT) + OPT$ for any machine g.

The claim is trivial for all poor machines since the load on a poor machine is less than $\sqrt{n}OPT$.

Otherwise, let S be the set of all active tasks that were assigned to machine g since its windfall time t_0. Let $j \in S$. Let $M(j)$ be the machines to which the job j could have been potentially assigned.

Since g was rich when j arrived, all machines in $M(j)$ must have been rich at that time. Moreover, all machines in $M(j) - \{g\}$ must have been rich at t_0, because otherwise a machine would have had windfall time later than g, and the algorithm would not have assigned j to g.

Let k be the number of machines to which any job in S could have been assigned, i.e., $k = \left| \bigcup_{j \in S} M(j) \right|$. Since they were all rich at t_0, Lemma 1 implies that $k \leq \sqrt{n}$.

Let q be the job that was assigned to g that made it rich at time t_0. Then:

$$\begin{aligned} l_g(t) &\leq \sqrt{n}L(t_0) + p_q + \sum_{j \in S} p_j \\ &\leq \sqrt{n}L(t) + OPT + kOPT \\ &\leq \sqrt{n}L(t) + OPT + \sqrt{n}OPT \\ &\leq (2\sqrt{n} + 1)OPT \quad ▮ \end{aligned}$$

It might be of some interest to notice that the bound cannot be improved beyond $\sqrt{n}$ by cleverly redefining *rich* machines as those with loads exceeding $\alpha L(t)$ since the coefficient of the last term in the above inequality would then be n/α.

Lower Bound

Theorem 8.5 *The competitive ratio for the* $1/\infty$ *scheduling problem with temporary tasks is* $\Omega(\sqrt{n})$.

Proof: The lower bound shall be proved for the following scenario:

- Call machines $1 \ldots n - \sqrt{n}$ *normal*, and the remaining $\sqrt{n}$ machines *special*.
- The jobs arrive in phases. $\sqrt{n}+1$ jobs arrive in phase i, each of which could be assigned to any of the special machines or normal machine i. There are $n-\sqrt{n}$ phases in all, from $i = 1, \ldots, n-\sqrt{n}$.

We will show that malicious adversary can choose a sequence such that the online algorithm is forced to have about $\sqrt{n}$ load, while the offline algorithm has OPT = 1. Figures 12 and 13 illustrate an example of this adversarial sequence.

In phase i, if the online algorithm schedules all of the $\sqrt{n}+1$ jobs on normal machine i, then i has a load of $\sqrt{n}+1$ and we are done. Otherwise, there exists at least one job p_i that online schedules on the special machines. The malicious adversary keeps p_i active "permanently" (until the end of the last phase), while the rest of the jobs are terminated at the end of phase i.

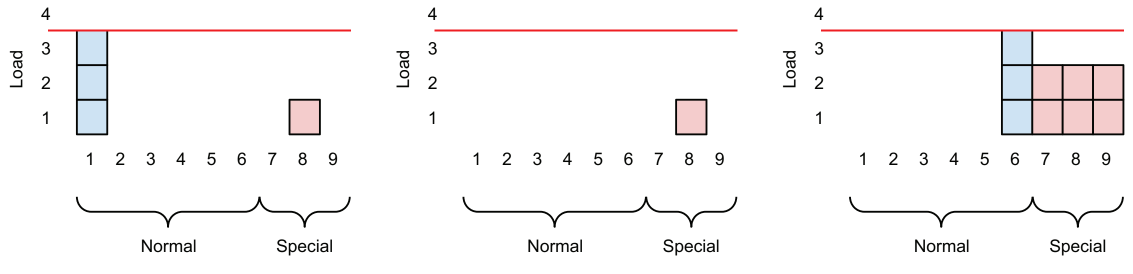
Therefore, at the end of phase $n-\sqrt{n}$, the number of jobs online has active on special machines is at least $n-\sqrt{n}$. Since there are only $\sqrt{n}$ special machines, there is at least one special machine with load at least $\sqrt{n}-1$.

In each phase, the offline algorithm takes the job p_i that the online puts on the special machine (and that remains stuck there "permanently"), and puts that job on the normal machine i. The other $\sqrt{n}$ jobs, which are removed at the end of the phase, are scheduled on the $\sqrt{n}$ special machines. So offline has load 1. Therefore, we have proved that the competitive ratio is $\Omega(\sqrt{n})$.

∎

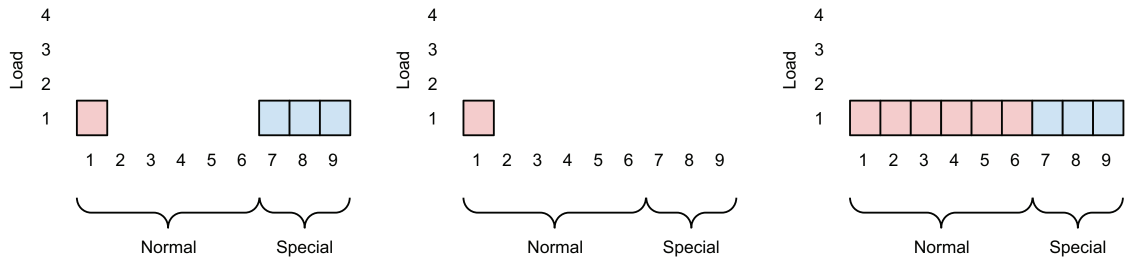
One had to quibble about the fact that the jobs lasted until the end of phase $n-\sqrt{n}$ and were not permanent to ensure that this lower bound implied a $O(\lg(nT))$ competitive ratio is impossible. In the construction above, we have $(n-\sqrt{n})(\sqrt{n}+1) \simeq n^{1.5}$ jobs in total, so the ratio of maximum to minimum job duration is $T \simeq n^{1.5}$. Therefore, it follows that a $O(\lg(nT))$ competitive ratio is impossible. So the competitive ratio here is much worse than the known duration case, where we had $\log(nT)$.

The lower bound is built on the notion that small local mistakes can have a very strong implication for global performance, therefore, if the job durations follow any reasonable distribution this lower bound completely falls apart.



(a) Allocation for phase 1 (b) After end of phase 1 (c) At end of last phase

Figure 12: Example of online algorithm in the lower bound instance. (a) In phase 1, $\sqrt{n}+1 = 4$ jobs must be placed on machine 1 or the special machines. At least one must be on a special machine, or we already exceed $\sqrt{n}$ load. (b) At the end of phase 1, one job which online placed on a special machine is kept "permanently", while the rest are terminated. (c) At the end of the last phase, at least $n - \sqrt{n} = 6$ jobs must be on the special machines, so there must be a machine with load at least $\sqrt{n} - 1 = 2$.



(a) Allocation for phase 1 (b) After end of phase 1 (c) At end of last phase

Figure 13: Example of optimal offline algorithm in the lower bound instance. (a) In phase i, the "permanent" job is placed on machine i, while the remaining jobs $\sqrt{n}$ jobs are placed on the $\sqrt{n}$ special machines. (b) At the end of phase i, all the non-"permanent" jobs which were placed on the special machines are terminated. (c) At the end of the last phase, offline has spread the "permanent" jobs across the normal machines, achieving maximum load 1.

8.4 Unrelated Machines with Known Durations

Here, we will present an algorithm for the situation where the machines are unrelated, but the durations of the jobs are known to the online algorithm. First, suppose that we know that all durations integers in the range $[1, T]$. Suppose that for every job, we have $0 \le \mathit{start} \le T$ and $\mathit{end} \le 2T$. Consider replicating every machine $2T$ times, one for each possible time. Then a job which is present at times $s, s+1, \cdots, t$ must be assigned to a collection of machines, one for each of these times and all corresponding to the same original machine.

This problem can be solved in much the same way as the online circuit routing problem and the previous unrelated machine load balancing problem: we define $\Phi = \sum_{i,t} a^{\ell_{i,t}(j)}$, where $\ell_{i,t}(j)$ is the load on the t-th copy of machine i after the first j jobs are assigned, and we always choose to assign a job to the machine which results in the smallest increase to Φ. Noting that there are nT machines here, the same analysis as before shows that this algorithm gives a competitive ratio of $O(\log(nT))$.

Now let us generalize this to the situation where durations are still bounded by T, but where start and end times can be arbitrary non-negative integers. We divide ("tile") the timeline into overlapping intervals $[0, 2T]$, $[T, 3T]$, $[2T, 4T], \cdots$. Then since any job has duration at most T, we can always find an interval which completely contains the job. Of course, the optimum offline algorithm can achieve L^* maximum load at all times, so in particular its maximum load on every interval is at most L^*. Moreover, the algorithm online algorithm can exceed the maximum load on any interval by a factor of $O(\log nT)$. Additionally, at any time there are no more than two "active" intervals, which can increase the maximum load by a factor of up to 2, so we still have an overall competitive ratio of $O(\log nT)$.

It is useful to note that the above approach directly translates to the congestion-minimization with known durations. In this model we are given a capacitated graph and each request i is for allocation of a certain capacity (bandwidth) p_i between two given nodes s_i and t_i *for some time interval* $[t_i, t'_i]$, where t_i and t'_i are integer start and stop times. As before, we first consider the case where all start and stop times are between 0 and some $2T$, and then use the "tiling" approach to extend to the general case.

Assume that the underlying graph is $G = (V, E)$. If all start and stop times are between 0 and $2T$, we can "time-expand" G and represent the problem as $2T$ copies of G, where each copy represents a specific time instance. The basic idea is that we satisfy request i by allocating some path P_i in all of the copies corresponding to the duration of request i.

More precisely, recall that for permanent requests (i.e. without durations), $\ell_e(i)$ was defined as

$$\ell_e(i) = \sum_{\substack{j \le i \\ e \in P_j}} p_e(j),$$

where $p_e(j) = p(j)/u(e)$, the fraction of edge e capacity taken by j-th request.

For the temporary case, the load depends on time, and hence we define

$$\ell_e(i, t) = \sum_{\substack{j \le i \\ e \in P_j \\ t \in [t_i, t'_i]}} p_e(j).$$

In other words, we sum up the relative load of requests that were using edge e and that were alive at time t.

In the congestion minimization model with permanent requests the request was routed along path

P_i that minimizes

$$W_i = \sum_{e \in P_i} \left(a^{\ell_e(i-1)+p_e(i)} - a^{\ell_e(i-1)} \right)$$

Similarly, in the temporary case we route along path P_i that minimizes:

$$W_i = \sum_{e \in P_i} \sum_{t \in [t_i, t'_i]} \left(a^{\ell_e(i-1,t)+p_e(i)} - a^{\ell_e(i-1,t)} \right)$$

It is straightforward to see that the proof goes through "as is", with only change being the total number of edges, which is now $2Tm$ instead of m. Hence we get $O(\log nT)$ competitive algorithm.

9 Off-line Routing

Since the results of the scheduling problems are going to applied to online routing problems, it might be instructive to consider off-line routing to understand the key ideas.

The statement of the routing problem in its simplest realization is as follows:

- There is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with a demands between certain pairs of nodes (s_i, t_i). The number of demand pairs is k, the number of vertices is n and the number of edges is m.
- One needs to find a routing paths for every pair of nodes for which have a demand which minimize the number of times a particular edge is in the various routing paths in the graph.

One can later complicate the issue by weighting the paths and also the edges. This problem can be approached as a multi-commodity flow problem. Let X_P be the variable associated with the path P in the graph $\mathcal{G}$ then

$$\begin{array}{rcll} & \min & W & \\ \sum_{P \in \mathcal{P}(s_q,t_q)} X_P & = & 1 & \forall q \\ \sum_{e \in p} X_P & \leq & W & \forall e \in \mathcal{E} \\ X_P & \geq & 0 & \forall P \in \mathcal{P} \end{array}$$

is the associated linear program. Although the number of constraints in the program is exponential in the parameters which makes it rather useless for any useful application, it has some pedagogical value. Since there exists at least one optimal basic solution to any linear program there is one solution with at most $k + m$ positive X_P's. Since the solution could have fractional X_P's it is not a solution for the routing problem as is. One way to generate a solution for the routing problem is to use path p with probability X_P [20]. If so, the second constraint in program will be met only in the expected value sense and we need to have some estimate on the probability that it will be violated. One can estimate it using the following Chernoff bound (stated without proof):

Lemma 9.1 *Let $X_1, X_2, \ldots, X_n$ be independent binary random variables, then:*

$$\begin{array}{rcll} I\!P\left(\sum_{i=1}^{n} X_i \geq (1+\delta) I\!E(\sum_i X_i)\right) & \leq & \left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{I\!E(\sum_i X_i)} & \\ & \simeq & e^{\frac{-\delta^2 I\!E(\sum_i X_i)}{4}} & \textit{for } \delta \textit{ sufficiently small} \end{array}$$

A similar result holds for $\geq$ replaced by $\leq$.

Let W^* be the optimal solution of the LP. Using the modification given above the expected number of paths through any edge is a sum of independent binary random variables with expectation less than W^*. If we want that the probability any edge exceeds $(1+\delta)W^*$ be less than ϵ then the above lemma implies that $\delta \simeq \sqrt{\frac{4 \ln(\frac{m}{\epsilon})}{W^*}}$. This implies that the total load $(1+\delta)W^*$ is $W^* + \sqrt{4W^* \ln(\frac{m}{\epsilon})}$. Therefore, if W^* is $\Omega(\lg(\frac{m}{\epsilon}))$ then the approximation is very good otherwise it is not. This puts us into a curious position where the quality of approximation depends on the solution on gets from the approximation.

10 Multicommodity Flow and Lagrangian Relaxation

Given source/sink pairs, we can compute routes by stating the problem as a multi-commodity flow, solving the relaxed LP, getting a *fractional* solution (corresponding to splitting the routes) and doing randomized rounding. This results in a good solution with high probability.

This approach, however, has several disadvantages. First, this procedure is relatively slow. Second, it is not clear how to use it to *improve* a given routing. Moreover, the procedure is completely centralized. In this lecture we will develop an iterative approach to computing routes (and solving multi-commodity flow problems). Among other properties, the iterated nature of this approach makes it more useful in an on-line setting.

10.1 Intuition behind the iterative approach

We will use an approach that can be viewed as a type of *Lagrangian Relaxation* procedure. Intuitively, the idea is to "move" some of the more complicated constraints into the objective function. Consider the problem where we are given two sets of constraints, $Ax \leq b$ and $\alpha x \leq \beta$. Assume we have an oracle that can minimize cx (for any constant c) subject to the constraints $Ax \leq b$. We can think of these as the "easy" constraints. We are responsible for satisfying the $\alpha x \leq \beta$ constraints some other way. We try to satisfy these "hard" constraints by making them part of the objective function (cx) that our oracle minimizes.

We do this by proceeding in rounds, where c in one round depends on α, x, and β from the previous round. In particular, we set $c = \phi(\alpha x_{prev}/\beta)$.

The idea is that if x changes little, minimizing cx is likely to move us closer to satisfying $\alpha x \leq \beta$. Unfortunately, we are not guaranteed that this iterative process converges. In order to guarantee polynomial time convergence, we will have to modify this approach.

Roughly speaking, our strategy will be to *move a little bit towards* $\tilde{x}$ that minimizes the new objective function, and choose this objective function so as to guarantee that we make a measurable improvement at every step. Our presentation will follow [18].

10.2 Approximation algorithm for multi-commodity flow

Terminology:

- $f_i(e)$: the flow of commodity i on edge e
- $f(e)$: the total flow on edge e
- $f'(e)$: the flow for the next round
- $u(e)$: the capacity of edge e
- λ : the *overflow*, the smallest number such that $f(e) \leq \lambda u(e)$ for all e

Remember that our goal is to minimize λ.

Multi-commodity flow has two kinds of constraints: conservation constraints and capacity constraints. If we ignore the capacity constraints, we're left with a shortest-paths (SP) problem. We put these constraints into $Ax \leq b$ and let SP be our oracle. SP will try to minimize cx. Thus $c(e)$ should be large if $f(e)/u(e)$ is large; on the other hand, we want c to be smooth to facilitate convergence. The definition we use (where we rename c to ℓ to capture the idea that this corresponds to edge lengths in a shortest paths context) is

$$\ell(e) = \frac{1}{u(e)} exp[\alpha f(e)/u(e)]$$

We are using a standard trick of estimating the maximum by summing over exponentials. Note that $\max_{\forall e} exp[\alpha(f(e)/u(e))] \leq \sum \ell(e)u(e) \leq m \times max_{\forall e} exp[\alpha(f(e)/u(e))]$, where m is the number of edges in the network.

Now that we've set up the inequalities, we have to decide how to iterate. There are many possibilities; the one presented below does not produce integer flows. We will defer the dealing with integrality until later. Given a flow $f(e)$, we use it to define $\ell(e)$ and run an SP algorithm to get $\tilde{f}$, the shortest paths for each commodity. We then move some fraction σ of the flow to $\tilde{f}$:

$$f' = \sigma \tilde{f} + (1 - \sigma) f$$

Our algorithm is then to pick a starting $f(e)$, use it to define $\ell(e)$, run an SP algorithm, move σ of the flow of each commodity from f_i to $\tilde{f}_i$ to get a new $f(e)$, and repeat. All we need to do is show that λ converges relatively quickly to the minimum possible solution.

10.3 Proof of Convergence

10.3.1 The approximation inequalities

More terminology:

$$\begin{aligned} C_i(e) &= f_i(e)\ell(e) \\ C_i &= \sum_e C_i(e) \end{aligned}$$

Let $\tilde{C}_i$ be demand$_i \cdot SP_i$, a lower bound on C_i (for the current ℓ). Likewise, we can bound C_i from above by $(\lambda u) \cdot \ell$, where u and ℓ are row and column vectors of capacities and costs, respectively.

This gives us our basic inequality. (Observe that it corresponds to weak LP duality.)

$$\lambda(u \cdot \ell) \geq \sum_{e,i} f_i(e)\ell_e = \sum_i C_i \geq \sum_i \tilde{C}_i$$

In particular, these inequalities are true for the optimal λ^*, so if we can find an f and corresponding λ for which both of these inequalities are relatively tight, then we have a good approximation of λ^*. In particular, we're done when we satisfy the following two inequalities:

$$(1 - \epsilon)\lambda(u \cdot \ell) \leq \sum_e f(e)\ell(e) \quad (16)$$

$$(1 - \epsilon)C_i \leq \sum_i \tilde{C}_i + \epsilon\lambda u\ell \quad (17)$$

Intuitively, **(16)**, which tries to keep every edge equally congested, is "easy"; **(17)**, which tries to keep most commodities on short paths, is "hard."

(I should point out that we're assuming ϵ is small enough that we can afford to be off by constant factors; in the above equations, for instance, we're assuming that $1/(1-\epsilon) \approx 1+\epsilon$.)

First we show that if α (in the definition of ℓ) is large enough, **(16)** is true. Intuitively, for large α uncongested edges have length 0. Formally, we consider the edges in two groups. First, consider an edge e such that $f(e) < (1-\epsilon)\lambda u(e)$. For this e, $\frac{\ell(e)u(e)}{\sum_e \ell(e)u(e)} < \frac{\exp(\alpha(1-\epsilon)\lambda)}{\exp(\alpha\lambda)} = e^{-\epsilon\alpha\lambda}$. Pick α so this quantity is less than ϵ/m. Then, summing over these edges only, $\lambda \sum \ell(e)u(e) < \epsilon\lambda(\ell \cdot u)$.

Now consider the other edges, where $f(e) \geq (1-\epsilon)\lambda u(e)$. Multiply both sides by $\ell(e)$ and sum over all e to get $\lambda(\ell \cdot u) \leq \frac{1}{1-\epsilon}\sum f(e)\ell(e)$. Taken together, the inequalities for each edge type give us **(16)**.

Instead of **(17)**, we're going to use

$$(1+\epsilon)c(i) - \tilde{C}_i \geq \epsilon\lambda(u \cdot \ell) \tag{18}$$

Observe that if the above inequality is not satisfied, while **(16)** is satisfied, we have $(1+O(\epsilon))$-optimum solution. Since we will choose α large enough for **(16)** to be always satisfied, we will be able to assume that **(18)** is always satisfied.

10.3.2 The potential function

We define the potential function $\Phi = \sum_e \ell(e)u(e)$. This is not the only choice, but notice that the gradient points in the direction of ℓ, so as we move flow onto the shortest path we're descending along the gradient — that is, we're optimizing with the gradient as our objective function!

We'll need the following approximation of $e^{x+\delta}$:

$$e^{x+\delta} \leq e^x + \delta e^x + \frac{\epsilon}{2}|\delta|e^x, \delta \leq \epsilon/4 \leq 1/4$$

(Remember: δ must be small!) Then we can bound the change in ℓ between iterations:

$$\ell'(e) - \ell(e) \leq \delta_e \ell(e) + \epsilon/2|\delta_e|\ell(e).$$

Let $\delta_e = \frac{\alpha}{u(e)}(f'(e) - f(e)) = \frac{\alpha\sigma}{u(e)}(\tilde{f}(e) - f(e))$, where the last equality comes from the construction of f'. This measures the percent change in the congestion of edge e. Then

$$\ell'(e) - \ell(e) \leq \sum_i \frac{\alpha\sigma}{u(e)}(\tilde{f}_i(e) - f_i(e))\ell(e) + \sum_i \frac{\epsilon}{2}\frac{\alpha\sigma}{u(e)}[\tilde{f}_i(e) + f_i(e)]\ell(e)$$

(Notice we got rid of the absolute value by using the fact $f_i(e) \geq 0$. This assumes edges are directed.)

Multiplying everything by $-u$ and doing some substitutions, we get

$$\Phi - \Phi' \geq \alpha\sigma \sum_i [C_i - \tilde{C}_i] - \frac{\epsilon}{2}\alpha\sigma \sum_i [C_i + \tilde{C}_i]$$

$C_i + \tilde{C}_i \leq 2C_i$, so the last term is at most $\epsilon\alpha\sigma \sum C_i$.

The assumption that (**18**) is true implies that we get a measurable decrease in the potential function:

$$\Phi - \Phi' \geq \alpha\sigma \cdot \epsilon\lambda(u \cdot \ell) = \Phi\alpha\lambda\epsilon\sigma.$$

We've conveniently forgotten that δ has to be small. In fact, $\delta < \epsilon/4$ implies $\sigma \frac{\tilde{f}-f}{u} \alpha < \epsilon/4$. For now, we will assume that there exists a value ρ, such that $(\tilde{f}(e) - f(e))/u(e) < \rho$ for all possible flows $\tilde{f}$. For example, if a commodity corresponds to a route, then each commodity should "fit" on a single path, i.e. $\rho \leq k$. For general multicommodity flow ρ can be unbounded. Techniques to overcome this problem were developed in [18, 19].

Using ρ, we can choose $\sigma \approx \epsilon/\alpha\rho$. This means that $\Phi - \Phi' \approx \frac{\lambda\epsilon^2}{\rho}\Phi$. That is, Φ is halved in $\rho\epsilon^{-2}/\lambda_{min}$ iterations (we use λ_{min} because, of course, λ is not constant between iterations).

In fact, we don't care about $\Delta\Phi$ as much as $\Delta\lambda$. But notice that if $\lambda_0/2 \leq \lambda \leq \lambda_0$, then $e^{\alpha\lambda_0/2} \leq \Phi \leq me^{\alpha\lambda_0}$. Thus, after $\alpha\lambda_0 + \log m \approx \alpha\lambda_0$ halvings of Φ, we've halved λ. This works out to $\alpha\lambda_0 \frac{\rho}{\lambda_0/2}\epsilon^{-2} \approx \alpha\rho\epsilon^{-2} \approx \epsilon^{-3}\lambda_0^{-1}\rho\log(m/\epsilon)$ iterations.

What value do we use for our starting λ? If we put every commodity on its shortest path, $\lambda \leq \max \tilde{f}(e)/u(e) = \rho$. This is enough to show convergence in polynomial time, assuming polynomial ρ.

10.4 ϵ scaling

The dependence on ϵ can be reduced to ϵ^{-2} by using ϵ-scaling. Assume we have a $1 + 2\epsilon$ approximation already. Then our bounds on Φ are $e^{\alpha\lambda^*} \leq \Phi \leq me^{\alpha\lambda^*(1+2\epsilon)}$. ($\lambda^*$ is the optimal λ.) Thus we only need $\epsilon\alpha\lambda_0$ "halving" steps, getting rid of an ϵ^{-1} in the time analysis. In return, we must repeat the ϵ scaling step, but since each time ϵ decreases by a factor of two, the times add up to approximately $2\epsilon_{final}^{-2}\lambda_0^{-1}\rho\log(m/\epsilon)$.

Remember that we were assuming ϵ is small (so that $1/(1-\epsilon) \approx 1+\epsilon$), so we can't start ϵ scaling until we have a good approximation already, about $\epsilon = 1/6$.

10.5 Integer flows

If we view each virtual circuit as a commodity, we want each commodity on only one path — that is, we want $\sigma = 1$. To achieve this, we will move only one commodity at a time, namely commodity i which maximizes $C_i - \tilde{C}_i$. $\sum C_i - \tilde{C}_i \geq \epsilon \sum C_i + \epsilon\lambda\Phi$, so $C_{i_{max}} - \tilde{C}_{i_{max}} \geq \frac{1}{k}\left[\epsilon \sum C_i + \epsilon\lambda\Phi\right]$. Since we only deal with a single commodity, ρ', which we define to be the maximum overflow of a single commodity, is the appropriate parameter. Note that since $\rho \approx k$, $\rho' \approx 1$. Now we let $\sigma = \min\{1, \frac{\epsilon}{4\alpha\rho'}\}$. This means we do $1/k$ as much work, but we move k times as much.

It's still possible to have $\sigma < 1$, but we just stop if that ever happens. The claim is that we still get a good approximation. The proof of this claim will be deferred to the next lecture. (What is the running time of the resulting algorithm ?)

11 Throughput Model

Up to now, we have been using the *Congestion Model*, where all requests must be routed and the goal is to minimize the maximum relative overload. In the *Throughput Model*, we are allowed to reject requests, but we must satisfy the capacity constraints on each edge. The goal is then to maximize the amount of requests routed.

11.1 Simple Example

First, we will consider a simplified situation, so we can get the intuition for the general case. We will consider:

- All edges have capacity k;
- The profit for any request is $p = 2^k = n$;
- All requests have rate 1.

11.1.1 The Algorithm

The algorithm is simply:

> For each request, calculate the shortest path (SP) with the costs $c_e(j) = 2^{\lambda_e(j)} - 1$ associated with each edge, where $\lambda_e(j) =\#$ of circuits routed through edge e right before request j is routed. If cost of SP $> p$ reject the request, otherwise rout through SP.

Claim 1: Capacity constraints are satisfied.

Proof: This is an obvious consequence of the definition of cost of an edge and the fact that all requests have rate 1.

Claim 2: Online Profit $\geq \frac{1}{2k+1}$-OPT profit. First, we lower bound the on-line profit by the cost and then upper bound the off-line profit by the cost.

(i): Total cost $\leq 2\times$ Total profit.

Proof: We will prove by induction on the number of requests. At the beginning both costs and profits are zero, so the result is obviously true. Suppose it is true up to j. For $j+1$, if the path assigned is P_{j+1} we have

$$\begin{aligned}
\sum_e [c_e(j+1) - c_e(j)] &= \sum_{e \in P_{j+1}} [c_e(j+1) - c_e(j)], && \text{since } c_e \text{ does not change on edges not used} \\
&= \sum_{e \in P_{j+1}} [2^{\lambda_e(j+1)} - 2^{\lambda_e(j)}] \\
&= \sum_{e \in P_{j+1}} [2^{\lambda_e(j)+1} - 2^{\lambda_e(j)}] \\
&= \sum_{e \in P_{j+1}} 2^{\lambda_e(j)} - |P_{j+1}| + |P_{j+1}| \\
&= \sum_{e \in P_{j+1}} (2^{\lambda_e(j)} - 1) + |P_{j+1}| \\
&\leq 2p.
\end{aligned}$$

where the inequality follows from the fact that request $j+1$ was routed and $\sum_{e \in P_{j+1}} (2^{\lambda_e(j)} - 1)$ is the cost of path P_{j+1} and the number of edges in P_{j+1} is bounded by the number of nodes in the graph $n = p$. QED.

(ii): Total missed profit $\leq k\times$ Total cost.

Proof Let $\mathcal{O}$ denote the set of job rejected by the on-line algorithm but used by OPT. Also, let l be the total number of jobs. Then, if $j \in \mathcal{O}$ we have

$$p \leq \sum_{e \in P_j} c_e(j).$$

Summing up over $\mathcal{O}$ we get

$$\begin{aligned}
\sum_{j \in \mathcal{O}} p &\leq \sum_{j \in \mathcal{O}} \sum_{e \in P_j} c_e(j) \\
&\leq \sum_e \sum_{j \in \mathcal{O}: e \in P_j} c_e(j) \\
&\leq \sum_e \sum_{j \in \mathcal{O}: e \in P_j} c_e(l+1), && \text{by the monotonicity of } c_e(\cdot) \\
&\leq k \sum_e c_e(l+1), && \text{since each edge can only route } k \text{ circuits. QED.}
\end{aligned}$$

Wrapping up: If $\mathcal{P}$ is the total profit achieved by OPT, and $\mathcal{A}$ is the set of requests routed by the on-line algorithm, we have

$$\begin{aligned}
\mathcal{P} &\leq \sum_{j \in \mathcal{O}} p + \sum_{j \in \mathcal{A}} p \\
&\leq k \sum_e c_e(\tau, l+1) + \sum_{j \in \mathcal{A}} p, && \text{by (ii)} \\
&\leq 2k \sum_{j \in \mathcal{A}} p + \sum_{j \in \mathcal{A}} p, && \text{by (i)} \\
&\leq (2k+1) \sum_{j \in \mathcal{A}} p. && \text{QED.}
\end{aligned}$$

11.2 General Case

Here, the presentation will follow [4].

11.2.1 Problem Statement and Definitions

The network is represented by a capacitated graph $G(V, E, u)$. $u(e)$ represents the capacity of each edge e. The input sequence is a collection of requests $\beta_1, \beta_2, \cdots, \beta_k$, where

$$\beta_i = (s_i, t_i, r_i(\tau), T^s(i), T^f(i), \rho(i))$$

and s_i, t_i are the origin and destination of request i; $r_i(\tau)$ is the traffic rate at time τ required by request i; $T^s(i), T^f(i)$ are the start and finish time of request i; $\rho(i)$ is the profit of request i;

We will define $r_i(\tau) = 0$ for $\tau \notin [T^s(i), T^f(i)]$. We also define $T(i) = T^f(i) - T^s(i)$, the duration of request i, $T = max_{1\leq j\leq k}T(j)$, the maximum duration of requests, and P_i to be the (s_i, t_i)-path assigned to request i. If request i was rejected, we set $P_i = \emptyset$.

The relative load on edge e just before considering the j-th request (i.e., before assigning a path to request j, but after assigning a path to request $j-1$) is defined by

$$\lambda_e(\tau, j) = \sum_{i<j} \frac{r_i(\tau)}{u(e)}.$$

Since we require that the capacity constraints be satisfied, we must have $\lambda_e(\tau, j) \leq 1, \forall \tau, e \in E, 1 \leq j \leq k$. The goal is to maximize the total profit, i.e, $\max \sum_{i:P_i\neq\emptyset} \rho(i)$.

The algorithm must be on-line in the sense that the routing or rejecting of β_i is done at time $T^s(i)$ without knowledge of future requests.

We will only consider profits that do not vary much from proportional to the bandwidth delay product, in the sense that for any connection β_j and $r_j(\tau) \neq 0$,

$$1 \leq \frac{1}{L}.\frac{\rho(i)}{r_j(\tau)T(j)} \leq F.$$

where L is the maximum number of hops that a rout can have. F is just a constant that bounds the amount of variation. Intuitively it is clear that we need some kind of restriction on the relative profits of the requests, otherwise we can always come up with a sequence of moderate profit requests that "exhausts" the resources followed by a giant profit request (say $2^{(\text{sum of all previous profits})}$) making the competitive ratio tend to 0.

We also define

$$\mu = 2LTF + 1$$

and assume that

$$r_j \leq \frac{\min_e\{u(e)\}}{\log \mu}.$$

This means that the requested rates are significantly smaller than the minimum capacity link in the network. This requirement is needed to get on-line algorithms with polylogarithmic competitive ratios [4].

11.2.2 The Admission Control and Routing Algorithm

For each edge e and time τ, define the cost function

$$c_e(\tau, j) = u(e)(\mu^{\lambda_e(\tau,j)} - 1).$$

The algorithm routes requests β_j on a path that is small w.r.t. a weighted average of these costs. For each edge $e \in P_j$, e's contribution to the cost is $\sum_\tau \frac{r_j(\tau)}{u(e)} c_e(\tau, j)$. If there exists a path P_j s.t.

$$\sum_{e \in P_j, \tau} \frac{r_j(\tau)}{u(e)} c_e(\tau, j) \leq \rho(j),$$

then this path is used to rout β_j otherwise β_j is rejected.

In the analysis of the algorithm we proved two claims:

Claim 1:The algorithm does not violate the capacity constraints.

Claim 2: The profit achieved by the algorithm is a $\frac{1}{2\log\mu+1}$-fraction of the profit achieved by the optimal off-line algorithm.

Claim 1: Intuitively, the capacity constraints are satisfied because when an edge is close to saturation, its cost is so high that the algorithm will not use it. Recall that requests have rates much smaller than $\min u(e)$ and thus a request cannot saturate an edge with small load.

Proof of Claim 1: Define $\mathcal{A} = \{i : P_i \neq \emptyset\}$, i.e., the set of routed requests. For the sake of contradiction, let β_j be the first request to cause an overflow. Then, for some $e \in E$ and $T^s(j) \leq \tau \leq T^f(j)$,

$$\begin{aligned} \lambda_e(\tau, j) &> 1 - \frac{r_j(\tau)}{u(e)} \\ &> 1 - \frac{1}{\log\mu}, \qquad \text{since } r_j(\tau) \leq \frac{u(e)}{\log\mu}. \end{aligned}$$

Then,

$$\begin{aligned} r_j(\tau)\frac{c_e(\tau,j)}{u(e)} &= r_j(\tau)(\mu^{\lambda_e(\tau,j)} - 1) \\ &\geq r_j(\tau)(\mu^{1-\frac{1}{\log\mu}} - 1) \\ &= r_j(\tau)(\frac{\mu}{2} - 1) = r_j(\tau)LTF \\ &\geq \rho(j), \qquad\qquad \text{since } \frac{1}{L}.\frac{\rho(i)}{r_j(\tau)T(j)} \leq F. \end{aligned}$$

This contradicts the assumption that β_j was routed. QED.

Claim 2: To prove the claim, we first prove a lower bound on the total profit of the on-line algorithm in terms of the link costs, then we prove an upper bound on the profit obtained by an optimal off-line algorithm on requests rejected by the on-line algorithm, in terms of the link costs, then, finally, we link all together to get the desired result.

Proof of Claim 2: The proof is a bit long, but the ideas are simple, and I'll try to make them clear whenever possible.

(i) The total profit achieved by the online algorithm is lower bounded by the cost of the links as follows:

$$2\log\mu \sum_{j\in\mathcal{A}} \rho(j) \geq \sum_{\tau}\sum_{e} c_e(\tau, k+1).$$

This should be intuitively clear, since we only routed requests, when the profit exceeded the cost. The $\log\mu$ constant comes from the requirement $r_j(\tau) \leq u(e)/\log\mu$.

The proof proceeds by induction on k. For $k = 0$ both sides are 0 and the inequality holds. Now, suppose it holds for $k = j$. We only need to show that the increase in cost is smaller than $2\rho(j)\log\mu$. Since rejected requests add 0 to both sides, we need only consider requests β_j that are routed. For these requests we need to show:

$$\sum_{\tau}\sum_{e}[c_e(\tau, j+1) - c_e(\tau, j)] \leq 2\rho(j)\log\mu.$$

For an edge $e \in P_j$, we have

$$\begin{aligned} c_e(\tau, j+1) - c_e(\tau, j) &= u(e)\mu^{\lambda_e(\tau,j)}(\mu^{\frac{r_j(\tau)}{u(e)}} - 1) \\ &\leq \mu^{\lambda_e(\tau,j)} r_j(\tau)\log\mu \\ &= (c_e(\tau, j)\frac{r_j(\tau)}{u(e)} + r_j(\tau))\log\mu. \end{aligned}$$

The inequality follows from $r_j(\tau) \leq u(e)/\log\mu$ and the fact that $2^x - 1 \leq x$ for $0 \leq x \leq 1$. The last equality follows from the definition of $c_e(\tau, j)$. Summing up on e and τ we get

$$\begin{aligned} \sum_{\tau}\sum_{e}[c_e(\tau, j+1) - c_e(\tau, j)] &\leq \log\mu(\rho(j) + \sum_{\tau}|P_j| r_j(\tau)) \\ &\leq \log\mu(\rho(j) + \sum_{\tau}\frac{\rho(j)}{T(j)}), \qquad \text{since } 1 \leq \frac{\rho(j)}{LT(j)r_j(\tau)} \text{ and } |P_j| \leq L, \\ &\leq 2\rho(j)\log\mu. \quad \text{QED.} \end{aligned}$$

(ii): Let $\mathcal{O}$ be the set of requests routed by OPT but not by on-line, so that the lost profit is $\sum_{j\in\mathcal{O}}\rho(j)$. Then, we will prove $\sum_{j\in\mathcal{O}}\rho(j) \leq \sum_{\tau}\sum_{e} c_e(\tau, k+1)$. If request $j \in \mathcal{O}$, then if P'_j is the path used by OPT,

$$\begin{aligned} \rho(j) &\leq \sum_{\tau}\sum_{e\in P'_j} c_e(\tau, j)\frac{r_j(\tau)}{u(e)} \\ &\leq \sum_{\tau}\sum_{e\in P'_j} c_e(\tau, k+1)\frac{r_j(\tau)}{u(e)}, \qquad \text{by the monotonicity of } c_e(\tau, \cdot). \end{aligned}$$

Summing over j, we get

$$\begin{aligned} \sum_{j\in\mathcal{O}}\rho(j) &\leq \sum_{\tau}\sum_{e} c_e(\tau, k+1) \sum_{j\in\mathcal{O}, e\in P'_j}\frac{r_j(\tau)}{u(e)} \\ &\leq \sum_{\tau}\sum_{e} c_e(\tau, k+1), \qquad \text{since OPT cannot exceed the capacity of edge } e. \text{ QED.} \end{aligned}$$

Wrapping up: If $\mathcal{P}$ is the profit achieved by OPT, we have

$$\begin{aligned}
\mathcal{P} &\leq \sum_{j\in\mathcal{O}} \rho(j) + \sum_{j\in\mathcal{A}} \rho(j) \\
&\leq \sum_{\tau}\sum_{e} c_e(\tau, k+1) + \sum_{j\in\mathcal{A}} \rho(j), \qquad \text{by (ii)} \\
&\leq 2\log\mu \sum_{j\in\mathcal{A}} \rho(j) + \sum_{j\in\mathcal{A}} \rho(j), \qquad \text{by (i)} \\
&\leq (2\log\mu + 1) \sum_{j\in\mathcal{A}} \rho(j). \qquad \text{QED.}
\end{aligned}$$

Since $\mu = 2LTF + 1$ and $L = O(n)$, we get that on-line achieves a fraction of $1/O(\log n)$ of the profit achieved by OPT.

12 Online Bipartite Matching Made Simple

In this section we review an online algorithm for bipartite matching. The algorithm appeared first in [16]. We will follow a simpler proof presented in [11].

Given bipartite graph $G = (U, V, E)$ (where vertices of one part is given in online fashion), and we want to match the maximum number of vertices in U with vertices in V. This is useful for many applications, e.g. online advertising, where you match keywords to bidders.

More formally, given $G(U, V, E)$, a node $u \in U$ arrives in an online fashion (note that vertices on V are fixed and given from beginning). As u arrives, all edges $(u, v) \in E$ are revealed. If v is not matched, and $(u, v) \in E$, we can choose to match it with u. The objective is to maximize the number of nodes matched. (Note that this means we want to *maximize* our competitive ratio). Below, we will adopt the convention of writing the u nodes on the left and the v nodes on the right.

First of all, note that the greedy deterministic algorithm (in which by arriving vertex u, we will match it to the first eligible node) will get $\frac{1}{2}$ competitive ratio. Firstly note that it will always gets $\frac{1}{2}$ competitive ratio, since the output of this algorithm is always a maximal matching, which always has the size half of the maximum matching. Also it will not get better than 1/2 competitive ratio, since assume bipartite graph $G(U, V, E)$, where $U = \{u_1, u_2\}$, $V = \{v_1, v_2\}$, and $E = \{(u_1, v_1), (u_1, v_2), (u_2, v_1)\}$. When vertex u_1 arrives online, the greedy algorithm will match it to v_1, and thus it cannot match u_2, however in the optimum algorithm both u_1 and u_2 will be matched.

The algorithm we're going to discuss is called *Ranking*. The procedure is as follows: initially, choose a random permutation σ for the fixed vertices V. We think of this as a ranking. Then, upon arrival of a vertex u on the left side, let $N'(u)$ be set of neighbors of u that have not been matched yet. If $N'(u) \neq \emptyset$, match u with the node in $N'(u)$ with the lowest rank according to σ.

We denote the output of the algorithm as $\mathrm{R}anking(G, \pi, \sigma)$ where π is the arrival order of u and σ is the permutation of the right side nodes.

We claim that ranking has a competitive ratio of $c = 1 - 1/e$. The basic strategy of our proof is to show that if Ranking fails to match a node, it must, in expectation, have matched most of the nodes before it. In order to understand what the algorithm does in expectation, we need to see how the results change when we change the permutation σ. To this end, we will prove three lemmas. We first make an essentially structural observation about what happens when we remove a node from the v nodes. Note that without loose of generality we can assume that G has a perfect matching. (If there were extra unmatched nodes, then this can only help online, because matching to an extra node doesn't take away any options from the online algorithm. This would increase the competitive ratio, which we don't care about because we're doing a lower bound). Let m^* be this perfect matching and by $v = m^*(u)$, we mean in the perfect matching vertex v is matched to vertex u.

Lemma 12.1 *Let x be a vertex in V and $H = G - \{x\}$. Let σ_H be the ranking of H induced by σ, and π_H be the ordering induced by π. If Ranking$(H, \pi_H, \sigma_H) \neq$ Ranking(G, π, σ), then they differ by a single* alternating path. *(An alternating path is a path with edges alternately belonging to Ranking(H, π_H, σ_H) and Ranking(G, π, σ).)*

Proof:

Let u be the node that x was matched to in G. Since we removed x, the edge (u, x) is gone from H. So if u is matched, it will be matched to some x' where $\sigma(x) \leq \sigma(x')$. This means that the node

[3]Presented by Sean Choi, Scribed by Anand Natarajan.

originally matched with x' is matched to someone else, and so on. This gives us an alternating path, going from u to x' to its new matching and so on. This is clear from figure14. ▮

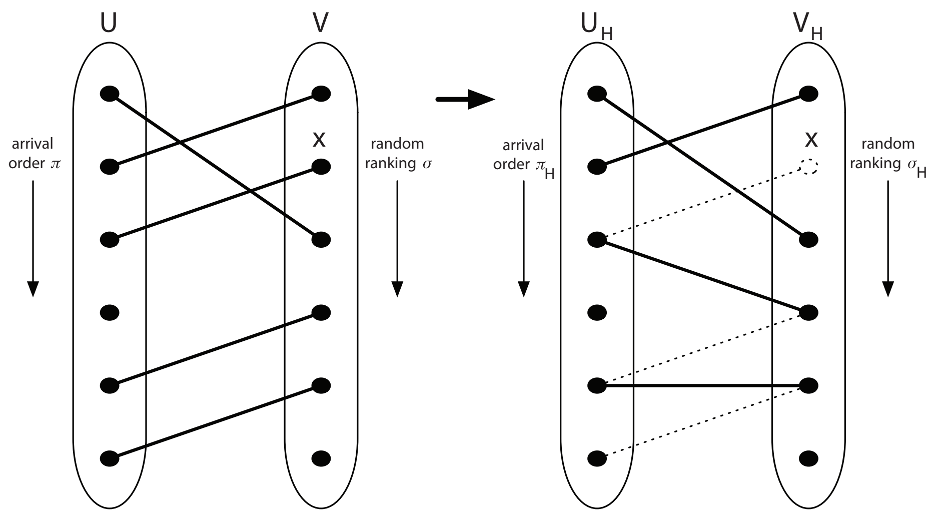


Figure 14: Illustration of lemma 12.1.

Additionally, knowing that our algorithm always matches to the lowest-ranked available vertex, we can say the following:

Lemma 12.2 *Fix $u \in U$, let $v = m^*(u)$. If v is not matched at all in Ranking(G, π, σ), then u is matched to v' where $\sigma(v') \leq t = \sigma(u)$.*

Proof: This is just because the algorithm greedily matches nodes to the lowest ranked available match. Since v is always free, the only way that u was not matched to v was that it was matched to something better. ▮

Now that we know what happens when we remove a right-hand vertex, let's examine what happens when we put it back in in a different slot in the ranking.

Lemma 12.3 *Let $u \in U$, $v \in m^*(u)$. Let σ' be a permutation and let σ_i be the permutation obtained from σ' by removing vertex v and putting it back so that its ranking is i. If v is not matched in Ranking(σ'), then for every i, u is matched by Ranking(σ_i) to a vertex v_i where $\sigma_i(v_i) \leq t = \sigma'(v)$.*

Proof: Let $m = \text{R}anking(\sigma'), m_i = \text{R}anking(\sigma_i)$. If v was not matched in σ', then we can just remove it to get a permutation σ'' without changing the behavior of the algorithm at all. Now, σ'' is equal to σ_i with the ith vertex removed. Thus, according to lemma 12.1, the matchings differ by an alternating path. If we recall the proof of lemma 12.1, the alternating path started at v and was monotone, i.e. it traversed vertices in V in order of increasing rank in σ_i. Now, let $v' = m(u)$. Then because the alternating path is monotone, $\sigma_i(v_i) \leq \sigma_i(v')$. Since we only perturbed the permutation by one vertex, $|\sigma_i(v') - \sigma'(v')| \leq 1$. Moreover, by lemma 12.2, $\sigma'(v') < t$. Therefore, $\sigma_i(v_i) \leq \sigma'(v') + 1 < t + 1$, so $\sigma_i(v_i) \leq t$. ▮

Now, we put all these together to show that if a vertex is not matched by online, then in expectation most of the previous vertices were matched, and thus we can expect good result from online algorithm.

Lemma 12.4 *Let x_t be the probability that a vertex of rank t is matched by the algorithm. Then* $1 - x_t \leq (1/n) \sum_{1 \leq s \leq t} x_s$.

This is consistent with the intuition description provided earlier: the RHS is the expected number of matchings of nodes ranked $\leq t$, and the LHS is the probability that the node with rank t is *not* matched.

To gain further intuition, think about the extreme case: x_t is very small. In this case, the lemma says all the previous vertices are matched. So there's a trade-off between matching this node and matching all its precedents. This is how we prove that in overall a lot of vertices will be matched: if so far a lot haven't been matched, then the next one is very likely to be matched.

Before we prove this lemma, let's see how to get from here to our final result. We can rewrite the lemma as $S_t(1 + 1/n) \geq 1 + S_{t-1}$ where $S_t = \sum_{1 \leq s \leq t} x_s$. Our performance is S_n. We want to consider the worst case, which is when S_n is minimum. That happens when all our inequalities are equalities. By induction, the worst we could possibly do is $S_t = \sum_{s=1}^{t}(1 - 1/(n+1))^s$:

$$\begin{aligned} S_{t-1} &= \sum_{s=1}^{t-1}(1 - 1/(n+1))^s \\ S_t &\geq \frac{n}{n+1}(1 + S_{t-1}) \\ &= \frac{n}{n+1} + \frac{n}{n+1}\sum_{s=1}^{t-1}(1 - 1/(n+1))^s \\ &= n\sum_{s=1}^{t}(1 - 1/(n+1))^s \end{aligned}$$

This is equal to $n(1 - 1/e)$ $n \to \infty$, and thus we get $1 - \frac{1}{e}$ competitive ratio.

Now all that remains is to prove the lemma.

Proof:

We first give an incorrect but intuitively correct proof. This was the original proof by Karp, Vazirani, and Vazirani, and it stood thus for 17 years before somebody noticed the error. That shows how subtle the issue is.

Let $v \in V$ whose rank is t. Let u be such that $v = m^*(u)$. Let $R_{t-1} \subset U$ be nodes matched by *Ranking* to V with rank $\leq t - 1$. $E[|R_{t-1}|] = \sum_{s=1}^{t-1} x_s$. If v is not matched, then by lemma 12.2, $u \in R_{t-1}$. If u and R_{t-1} were independent, then $P[u \in R_{t-1}] = |R_{t-1}|/n = \sum_{s=1}^{t-1} x_s(1/n)$ which would prove our lemma. However, our procedure for choosing u starts by choosing t: given a permutation, choosing t corresponds to a vertex which then gives us u. So u depends on t and σ. Thus, u is not independent from t and thus from R_{t-1}.

We correct this by using lemma 12.3, where we removed the dependence on v by removing v from σ'. Given σ, let σ' be a permutation obtained by choosing $v \in V$ at random and moving it back, so that its rank is t. Consider R*anking*(σ'). Let u be such that $v = m^*(u)$. By lemma 12.3, if v is not matched by R*anking*(σ'), then u is matched by R*anking*(σ) to a vertex $\tilde{v}$ such that $\sigma(\tilde{v}) \leq t$. This is equivalent to saying that $u \in R_t$. Now u is independent of σ and hence R_t, so the argument presented above works. ∎

To put this all in perspective: the simple (deterministic) algorithm already had a competitive ratio of $1/2$, so we did all this work to improve that up to $1 - 1/e \approx 0.632$.

References

[1] Susanne Albers. On randomized online scheduling. In *STOC*, pages 134–143, 2002.

[2] N. Alon and Y. Azar. On-line Steiner trees in the Euclidian plane. *Discr. & Comp. Geometry*, 10(2):113–121, 1993.

[3] J. Aspnes, Y. Azar, A. Fiat, S. Plotkin, and O. Waarts. On-line machine scheduling with applications to load balancing and virtual circuit routing. *J. Assoc. Comput. Mach.*, 44(3):486–504, 1997.

[4] 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.

[5] Baruch Awerbuch and Yossi Azar. Buy-at-bulk network design. In *Proceedings of the 38th IEEE Symposium on Foundations of Computer Science*, pages 542–547, 1997.

[6] Baruch Awerbuch, Yossi Azar, Serge A. Plotkin, and Orli Waarts. Competitive routing of virtual circuits with unknown duration. *J. Comput. Syst. Sci.*, 62(3):385–397, 2001.

[7] Y. Azar, B. Kalyanasundaram, S. Plotkin, K. Pruhs, and O. Waarts. On-line load balancing of temporary tasks. *J. Algorithms*, 22(1):93–110, 1997.

[8] Y. Azar, S. Naor, and R. Rom. The competitiveness of on-line assignments. *Journal of Algorithms*, 18:221–237, 1995.

[9] Yossi Azar, Andrei Z. Broder, and Anna R. Karlin. On-line load balancing (extended abstract). In *FOCS*, pages 218–225, 1992.

[10] 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.

[11] Benjamin Birnbaum and Claire Mathieu. On-line bipartite matching made simple. *SIGACT News*, 39(1):80–87, March 2008.

[12] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms. *Journal of Algorithms*, 12:685–699, 1991.

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

[14] D. Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8:272–314, 1974.

[15] D. Karger, S. Phillips, and E. Torng. A better algorithm for an ancient scheduling problem. In *Proc. 5th ACM-SIAM Symposium on Discrete Algorithms*, 1994.

[16] R. M. Karp, U. V. Vazirani, and V. V. Vazirani. An optimal algorithm for on-line bipartite matching. In *Proceedings of the twenty-second annual ACM symposium on Theory of computing*, STOC '90, pages 352–358, New York, NY, USA, 1990. ACM.

[17] 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. *SIAM Journal on Computing*, 23(3):466–487, June 1994.

[18] T. Leighton, F. Makedon, S. Plotkin, C. Stein, É. Tardos, and S. Tragoudas. Fast approximation algorithms for multicommodity flow problems. *J. Comp. and Syst. Sci.*, 50:228–243, 1995. (Invited paper).

[19] S. Plotkin, D. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math of Oper. Research*, 20(2):257–301, 1995.

[20] Prabhakar Raghavan and Clark D. Thompson. Randomized rounding: a technique for provably good algorithms and algorithmic proofs. *Combinatorica*, 7(4):365–374, 1987.

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

[22] D. Sleator and R. Tarjan. Amortized efficiency of list update and paging rules. *Communications of ACM*, 28:202–208, 1985.

[23] A. Yao. New algorithms for bin packing. *Journal of ACM*, 27:207–227, 1980.