# CS369N: Beyond Worst-Case Analysis
# Lecture #8: Resource Augmentation[*]

Tim Roughgarden[†]

November 28, 2009

## 1 Origins: Online Paging Revisited

We've finished our discussion of models of data and now conclude the course with two lectures on novel ways of proving relative approximation guarantees. This lecture is about *resource augmentation*, where the idea to compare a protagonist (like your algorithm) that is endowed with "extra resources" to an all-powerful opponent that is handicapped by "less resources". Subject to this, we use standard worst-case analysis (with no model of the data).

Resource augmentation was perhaps first used by Sleator and Tarjan [?] in the context of online paging, a problem that we discussed at length in Lecture #2.[1] Recall the proof the the LRU (Least Recently Used) algorithm, among others, is a $k$-competitive online algorithm for the paging problem.

*Proof:* Consider an arbitrary request sequence $\sigma$. We can break $\sigma$ in *blocks* $\sigma_1, \sigma_2, \ldots, \sigma_b$ where $\sigma_1$ is defined as maximal prefix of $\sigma$ in which only $k$ distinct pages are requested; $\sigma_2$ starts immediately after and is defined to be maximal subject to only $k$ distinct pages being requested (ignoring what was requested in $\sigma_1$); and so on.

The first important point is that LRU faults at most $k$ times within a single block — at most one per page requested in the block. The reason is that once a page is brought into the cache, it won't be evicted until $k$ other distinct pages get referenced, which can't happen until the following block. Thus LRU incurs at most $bk$ page faults, where $b$ is the number of blocks. See Figure **??**(a).

Second, we claim that an optimal offline algorithm must incur at least $b - 1$ page faults. To see this, consider the first block plus the first request of the second block. Since $\sigma_1$ is maximal, this represents requests for $k + 1$ distinct page requests, and no algorithm can serve them all without a page fault. Similarly, suppose the first request of $\sigma_2$ is the page $p$.

†Department of Computer Science, Stanford University, 462 Gates Building, 353 Serra Mall, Stanford, CA 94305. Email: `tim@cs.stanford.edu`.

[1]The actual phrase "resource augmentation" is from [?].

After-wards, the cache can only contain $k-1$ pages other than $p$. But by maximality of $\sigma_2$, the rest of $\sigma_2$ and the first request of $\sigma_3$ contain requests for $k$ distinct pages other than $p$; these cannot all be served without incurring another page fault. And so on, resulting in at least $b-1$ cache misses. See Figure **??**(a). The theorem follows. ■

More generally, suppose the optimal offline algorithm has a cache with size only $h \leq k$. Then, the third paragraph of the proof immediately shows that in each "shifted block" the optimal algorithm incurs at least $k-(h-1) = k-h+1$ page faults (there are $k$ requests for pages other than $p$ in the shifted block and only $h-1$ such pages in the cache at its start). Thus:

**Theorem 1.1 (Resource Augmentation Bound for LRU [?])** *The competitive ratio of the LRU algorithm for online paging is at most $k/(k-h+1)$ against an offline adversary with a cache of size $h \leq k$.*

For example, if $h \approx \frac{k}{2}$, then LRU is *2-competitive*, a seemingly far stronger guarantee than our original (tight) $k$-competitive upper bound.

## 2 Interpretations

### 2.1 A Two-Step Approach to System Design

Some obvious questions are: what does the guarantee in Theorem 1.1 mean? Should you be impressed by a resource augmentation guarantee like this? The concern is that the comparisons is "apples vs. oranges" — sure the offline algorithm knows all of the future requests, but it's artificially hobbled by a small cache.

The first justification is not mathematical but nevertheless useful in many contexts. If we adopt the philosophy that the point of rigorous guarantees for algorithms is to give good advice about how to solve problems and build systems, then resource augmentation bounds offer a compelling two-step approach.

1. The first step is to estimate the amount of resources (e.g., cache size) that guarantees acceptable performance (e.g., page fault rate) assuming an optimal algorithm.[2] It is presumably simpler to solve this problem than to reason simultaneously about the cache size and paging algorithm design decisions.

2. The second step is to scale up the resources to realize the resource augmentation bound — for example, to double the cache size and invoke Theorem 1.1 to guarantee acceptable performance under (say) the LRU algorithm.

[2]As discussed in Lecture #3, when proving approximation bounds, we have in mind the problem instances where the "optimal performance" is actually good in some absolute sense!

## 2.2 Loosely Competitive Online Algorithms

The second interpretation of Theorem 1.1 is mathematical and less well known. Young [**?**] proved (a generalization of) the following guarantee, which we state informally now and make precise in due time.

**Informal Theorem [?]:** *For every request sequence $\sigma$, the LRU algorithm is $O(1)$-competitive on $\sigma$* for most cache sizes *$k$.*

This is a compelling guarantee because we don't expect "real data" to be adversarially tailored to the cache size — typically it will be independent of the cache size.[3]

There is simple and accurate intuition behind this informal theorem, as follows. Consider a request sequence $\sigma$ and a cache size $k$. One case is that the number of page faults of LRU is roughly the same (within a factor of 2, say) with the cache sizes $k$ and $2k$. In that case, Theorem 1.1 immediately implies that LRU has a good competitive ratio (in the traditional sense) when the cache size is $k$. In the second case, LRU's performance is changing rapidly if one supplements the cache with extra pages. But since there is a bound on the maximum fluctuation of LRU's performance (between no page faults and faulting every time step), its performance can only change rapidly for bounded number of different cache sizes.

More precisely, fix a request sequence $\sigma$ and let $b$ be a parameter. Let $\mathrm{cost}(A, k, z)$ denote the number of page faults incurred by the paging algorithm $A$ with a cache size of $k$ on the input $z$. Theorem 1.1 and the previous paragraph imply that, for every cache size $k$, either:

$$\mathrm{cost}(LRU, k+b, \sigma) < \frac{1}{2} \cdot \mathrm{cost}(LRU, k, \sigma) \tag{1}$$

or

$$\mathrm{cost}(LRU, k, \sigma) \leq 2 \cdot \frac{k+b}{b+1} \cdot \mathrm{cost}(OPT, k, \sigma), \tag{2}$$

where we are invoking Theorem 1.1 with $k+b$ and $k$ playing the roles of $k$ and $h$, respectively.

Call a cache size $k$ *bad* if (1) holds. Consider the set of bad cache sizes (Figure **??**); for every such size, adding $b$ extra pages to the cache decreases the cost of LRU on $\sigma$ by at least a factor of 2. If there are at least $\ell$ bad cache sizes between 1 and $k-b$ for some $k$, then we can find at least $\ell/b$ bad cache sizes in this interval that are each at least $b$ apart (the worst case is when bad cache sizes appear in clusters of $b$ consecutive sizes). In this case,

$$\mathrm{cost}(LRU, k, \sigma) < 2^{-\ell/b} \cdot \mathrm{cost}(LRU, 1, \sigma). \tag{3}$$

(Note that we are also using the fact that $\mathrm{cost}(LRU, t, \sigma)$ is nondecreasing in $t$, which you should prove as an easy exercise.)

Thus, once

$$\ell \geq b \cdot \log_2 \tfrac{1}{\epsilon}, \tag{4}$$

we have

$$\mathrm{cost}(LRU, k, \sigma) \leq \epsilon \cdot |\sigma|,$$

[3] In highly optimized applications the request sequence might depend on the cache size, but in a helpful, rather than harmful, way.

where $|\sigma|$ is the length of the request sequence $\sigma$. Young [**?**] makes the compelling argument that if $\epsilon$ is sufficiently small (less than the access time to fast memory, say), then LRU's performance is good in an absolute sense and we could care less about its competitive ratio.

Here is the precise statement of Young's theorem.

**Theorem 2.1 (LRU is Loosely Competitive [?])** *For every $\epsilon, \delta > 0$ and positive integer $n$, for every request sequence $\sigma$, for all but a $\delta$ fraction of the cache sizes $k$ in $\{1, 2, \ldots, n\}$, the LRU algorithm satisfies either:*

*(1) $cost(LRU, k, \sigma) = O(\frac{1}{\delta} \log \frac{1}{\epsilon}) \cdot cost(OPT, k, \sigma)$; or*

*(2) $cost(LRU, k, \sigma) \leq \epsilon \cdot |\sigma|$.*

Theorem 2.1 says that, for a fixed request sequence $\sigma$, very cache size $k$ falls into one of three cases. In the first case, LRU with cache size $k$ is competitive with OPT in the usual (non-resource augmentation) sense. In the second case, LRU has excellent performance in an absolute case. In the third case neither of these two goods events occurs, but fortunately this occurs for only a $\delta$ fraction of the possible cache sizes.

We have essentially already proved Theorem 2.1. We want $\delta n$ bad cache sizes between 1 and some number $t$ to force the condition that $\text{cost}(LRU, k, \sigma) \leq \epsilon|\sigma|$ for all cache sizes $k \geq t$, so we take $\ell = \delta n$; using (4), this forces $b = \delta n / \log_2 \epsilon^{-1}$. Then, for all but $\ell = \delta n$ cache sizes $k$, either $\text{cost}(LRU, k, \sigma) \leq \epsilon \cdot |\sigma|$ or (by (2)) the LRU algorithm has competitive ratio

$$\frac{2(k+b)}{b+1} \leq \frac{2(n+b)}{b+1} = \Theta\left(\frac{1}{\delta} \log \frac{1}{\epsilon}\right).$$

In [**?**] this guarantee is phrased as: LRU is $(\epsilon, \delta)$-*loosely* $O(\frac{1}{\delta} \log \frac{1}{\epsilon})$-*competitive.*

## 3 Resource Augmentation in Selfish Routing

### 3.1 A Motivating Example

Our next example of a resource augmentation guarantee is for *selfish routing*, and it is due to Roughgarden and Tardos [**?**]. In selfish routing, we consider a directed flow network $G = (V, E)$, with $r$ units of traffic traveling from a source $s$ to a sink $t$. Each edge $e$ of the network has a flow-dependent cost function $c_e(x)$. For example, in the network in Figure **??**, the top edge has a constant cost function $c(x) = 1$, while the cost to traffic on bottom edge equals the amount of flow $x$ on the edge.

The key approximation concept in selfish routing networks is the *price of anarchy*, which as usual is defined as the ratio between a realizable protagonist and a hypothetical benchmark. The optimal solution is the fractional $s$-$t$ flow that routes the $r$ units of traffic to minimize the cost $\sum_e c_e(f_e) f_e$, where $f_e$ denotes the amount of flow on edge $e$. For example, in Figure **??**, the optimal flow splits traffic evenly between the two paths, for a cost of $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$.

In the *equilibrium flow*, all traffic is routed on shortest paths, where the length of a path $P$ is the flow-dependent quantity $\sum_{e \in P} c_e(f_e)$. The optimal flow in Figure **??** is not an equilibrium flow: traffic on the top path is not routed on a shortest path, and hence would want to revise its route. In the only equilibrium flow in Figure **??**, all traffic uses the bottom path for an overall cost of $0 \cdot 1 + 1 \cdot 1 = 1$.

The price of anarchy of a selfish routing network is defined as the ratio between the cost of an equilibrium flow and an optimal flow.[4] In the network in Figure **??**, the price of anarchy is $4/3$.

An interesting research goal is to identify selfish routing networks in which is price of anarchy is close to 1 — this indicates that the decentralized optimization performed by selfish network users performs almost as well as centralized and coordinated optimization. Unfortunately, with general (nonlinear) cost functions, the price of anarchy can be arbitrarily large. To see this, replace the cost function on the bottom edge in Figure **??** by $c(x) = x^d$ for $d$ large. The equilibrium flow remains the same, with all selfish traffic using the bottom edge for an overall cost of 1. The optimal flow improves with $d$, however: splitting the traffic $1-\epsilon$ on the bottom edge and $\epsilon$ on the top yields a flow with cost $\epsilon + (1-\epsilon)^{d+1}$. As $d \uparrow \infty$ and $\epsilon \downarrow 0$ appropriately, this cost tends to zero, and hence the price of anarchy tends to infinity.

## 3.2 A Resource Augmentation Guarantee

Despite the negative example above, a very general resource augmentation guarantee holds in selfish routing networks.

**Theorem 3.1 ([?])** *For every network $G$ with continuous and nondecreasing cost functions, for every traffic rate $r$ and $\delta > 0$,*

$$\textit{equilibrium flow cost at traffic rate } r \leq \frac{1}{\delta} \cdot \textit{optimal flow cost at traffic rate } (1+\delta)r.$$

For example, consider the network in Figure **??** with $r = \delta = 1$. We know that the equilibrium flow cost at traffic rate 1 is 1. The optimal flow can route one unit cheaply (as we've seen), but then the network is "clogged up" and it has no choice but to incur one unit of cost on the second unit of flow (the best it can do is route it on the top edge). Thus the cost of an optimal flow at double the traffic exceeds that of the original equilibrium flow.

Before proving Theorem 3.1 we mention two corollaries. First, the theorem can reformulate to hew more closely to the spirit of "resource augmentation", as follows: the cost of an equilibrium flow in a network with faster links compares favorable with that of an optimal flow in the original network. For example, the $\delta = 1$ case corresponds to augmenting a cost function $c(x)$ to the cost function $c(x/2)/2$. For the cost function $1/(u-x)$, which is a natural model of an edge with capacity $u$, this transformation yields $1/(2u-x)$, which corresponds to doubling the edge capacity. The proofs of these statements are fairly easy and mechanical, and we leave them as exercises.

[4]We won't prove it, but it turns out that the equilibrium flow cost is uniquely defined in every selfish routing network with continuous and nondecreasing edge cost functions.

Second, Theorem 3.1 leads to a "loose competitiveness" analogous to Theorem 2.1, which roughly states that, in every network, the price of anarchy is much better for "most" traffic rates than for a worst-case traffic rate [**?**]. You are asked to provide the details in HW #3.

*Proof of Theorem 3.1:* Fix $G$, $r$, and $\delta$. Let $f$ denote an equilibrium flow at traffic rate $r$ and $f^*$ an optimal flow at rate $r(1+\delta)$. The key trick is replace, for the sake of analysis, each cost function $c_e(x)$ (Figure **??**(a)) by the larger cost function $\hat{c}_e(x) = \max\{c_e(x), c_e(f_e)\}$ (Figure **??**(b)). With the artificial cost functions $\hat{c}_e$, edge costs are always as large as if the equilibrium flow $f$ has already been routed in the network.

Recall that in an equilibrium flow, all traffic is routed on paths with the minimum length. In $f$, call this common length $L$. Then the overall cost of $f$ is $r \cdot L$.

As for the optimal flow $f^*$, first consider its cost with the artificial cost functions $\hat{c}_e$. With these cost functions, even before any flow has been routed, every $s$-$t$ path has cost at least $L$. Since $f^*$ routes at least $(1+\delta)r$ units flow on paths with (artificial) cost at least $L$, its total (artificial) cost with respect to the $\hat{c}_e$'s is at least $(1+\delta)rL$.

We can complete the proof by showing that the artificial cost of $f^*$ (with respect to the $\hat{c}_e$'s) exceeds its real cost (with respect to the $c_e$'s) by at most $rL$, the equilibrium flow cost. This follows from summing over an edge-by-edge bound:

$$\hat{c}_e(f_e^*)f_e^* - c_e(f_e^*)f_e^* \le c_e(f_e)f_e. \tag{5}$$

To see why (5) holds, simply note that $\bar{c}_e(x) - c_e(x)$ is zero for $x \ge f_e$ and bounded above by $c_e(f_e)$ for $x < f_e$. Pictorially, the error is the shaded region in Figure **??**(b), and the worst-case width and height of this region are $f_e$ and $c_e(f_e)$, respectively. ■

## 4 Resource Augmentation in Scheduling

Resource augmentation bounds have found the widest number of applications in scheduling problems, and we conclude with a paradigmatic analysis of Kalyanasundaram and Pruhs [**?**].

### 4.1 The Model

Consider a single machine and jobs that arrive online. Each job has a release data $r_j$, at which time the algorithm becomes aware of it, and a processing time $p_j$, which the amount of time the job needs from the machine before it is complete. We assume that preemption is allowed, meaning that a job can be stopped mid-execution and restarted from the same point (with no loss) at a subsequent time.

We consider the natural objective of minimizing the total flow (or response) time:

$$\sum_{\text{jobs } j} (C_j - r_j),$$

where $C_j$ denotes the completion time of job $j$. A folklore result is that this is an easy problem: the *shortest remaining processing time (SRPT)* algorithm, which always works on the job that is closest to completion (preempting jobs as needed), is an optimal algorithm. We leave the proof as an easy exercise. Thus an online greedy algorithm is guaranteed to solve the problem optimally.

The problem becomes difficult if we insist on a *non-clairvoyant* scheduling algorithm, meaning one that is unaware of the processing times $p_j$. It is easy to imagine real-world scenarios where this constraint it important. However, it is known that no non-clairvoyant online algorithm has a good competitive ratio with respect to the optimal algorithm SRPT. Also, the algorithms that have non-trivial (but still bad) competitive ratios for the problem tend to be unnatural. These discouraging facts suggest that a resource augmentation analysis might be more useful and illuminating.

We compare the total flow time of the SRPT algorithm with a unit-speed machine to that of a non-clairvoyant algorithm with a machine with speed $1 + \epsilon$ (so that a job with processing time $p_j$ needs $p_j/(1 + \epsilon)$ units of machine time to complete). We will prove the following resource augmentation guarantee.

**Theorem 4.1** **([?])** *There is a simple non-clairvoyant scheduling algorithm that, for every* $\epsilon > 0$*, is* $(1 + \frac{1}{\epsilon})$*-competitive in the above sense.*

## 4.2 The Balance Algorithm

The "simple algorithm" of Theorem 4.1 is called the Balance Algorithm, and it works as follows. Observe that the Balance Algorithm is non-clairvoyant.

---

**Input**: an online sequence of jobs with unknown processing times.

1. For every job $j$ seen so far, keep track of its *weight* $w(j)$, defined as the amount of $j$'s processing time that's been complete so far.

2. At every time $t$, process all of the uncompleted jobs that have the minimum value of $w(j)$. If there is a tie among multiple jobs, process them all in parallel (i.e., round-robin with an arbitrarily fast rotation among them).

Figure 1: The Balance Algorithm.

---

We will give a "time step-by time step" analysis. Call a job *active at time t* if it has been released ($t \geq r_j$) but not completed ($t < C_j$). We show the following:

$$\begin{array}{l}\text{\# of active jobs at time } t\\ \text{under Balance (speed } 1+\epsilon)\end{array} \quad \leq \quad \left(1 + \frac{1}{\epsilon}\right) \times \begin{array}{l}\text{\# of active jobs at time } t\\ \text{under SRPT (unit speed)}\end{array} \tag{6}$$

Note that if $k$ jobs are active at a time $t$, then their marginal contribution at this time to the eventual flow time is $k\,dt$. Thus, integrating (6) over time implies Theorem 4.1.

Since the total flow time is the integral (over $t$) of the number of active jobs at time $t$, we can now see more clearly why SRPT is a good idea — it decreases the number of uncompleted jobs in the system as fast as possible. This also shows why the Balance Algorithm is not optimal — it might hedge its bets by processing a number of jobs in parallel, most of which have big processing times, rather than quickly disposing of a job that has a small processing time.

## 4.3 The Analysis

We now prove (6), in several steps. Fix an arbitrary input, value of $\epsilon$, and time $t$. Let $X$ and $Y$ denote the active jobs at time $t$ under the Balance Algorithm with a $(1+\epsilon)$-speed machine and SRPT under a unit-speed machine, respectively. Our goal is to show that $|X| \leq (1+\frac{1}{\epsilon})|Y|$.

The high-level idea is an exchange argument — since Balance never idles unnecessarily, whatever time is should have using to complete jobs of $X$ were used instead on other jobs, which are presumably in $Y$ — but the details are non-trivial and require several ideas. For example, it does not seem easy to directly charge uncompleted work in $X$ to the jobs of $Y$ — perhaps instead of working on something in $X$ the Balance algorithm was working on a job that was completed by both Balance and SRPT by time $t$ (i.e., a job in neither $X$ or nor $Y$). Addressing this problems appears to require considering cascaded sequences of exchanges; we next introduce a combinatorial object to reason (indirectly) about such sequences.

**Exchange Trees.** Rename the jobs of $X \setminus Y$ as $\{1, 2, \ldots, k\}$, with

$$t \geq r_1 \geq r_2 \geq \cdots \geq r_k \geq 0. \tag{7}$$

For each fixed $i \in X \setminus Y$ we define a breadth-first-search-esque *exchange tree* $T_i$. Nodes will correspond to jobs (which may or may not be in $X \setminus Y$). Level 0 of $T_i$ is the job $i$. Proceeding inductively, there is an edge from a job $j_{\ell-1}$ at level $\ell - 1$ to a job $j_\ell$ at level $\ell$ if and only if: (i) the job$j_\ell$ does not appear in any previous levels; and (ii) $j_\ell$ was processed by the Balance algorithm instead of $j_{\ell-1}$ at some time at which $j_{\ell_1}$ was active (at or before time $t$).

**Example 4.2** Consider the schedule shown in Figure **??**(a). At time 0 jobs 1 and 2 are released, and both have very large processing times. These are processed in parallel until jobs 3 and 4 are released simultaneously; at this point the Balance Algorithm preempts the first two jobs and starts processing the second two jobs in parallel. Jobs 3 and 4 have equal and short processing times, so both complete quickly and then the Balance Algorithm resumes processing jobs 1 and 2 in parallel. Then the job $i$ is released. The Balance Algorithm works solely on job $i$ until its has been processed as much as the first two jobs; at that point, it processing all three remaining jobs (1, 2, and $i$) in parallel. This is the state of the system at time $t$.

What does the exchange tree $T_i$ look like? At level 0 is the job $i$. Jobs 1 and 2 are in level 1, because when the Balance Algorithm is processing 1, 2, and $i$ in parallel at the end of the schedule, jobs 1 and 2 are taking time away from job $i$. Jobs 3 and 4 never directly interfere with job $i$, so they are not at level 1 of the tree $T_i$. Jobs 3 and 4 preempted jobs 1 and 2, however, so both appear in level 2 of $T_i$ with all arcs between $\{1, 2\}$ and $\{3, 4\}$ present. See Figure **??**(b).

There are four main steps in proving the bound (6). Define the *lifetime* of a job $j$ as the interval $[r_j, \min\{C_j, t_t\}]$, where the completion time $C_j$ is with respect to the Balance Algorithm.

**Step 1.** We first claim that the edge $a \to b$ appears in $T_i$ only if the lifetimes of jobs $a$ and $b$ overlap. This is by definition: for this edge to exist, job $b$ must be processed (and hence active) by the Balance Algorithm at a time $s \le t$ at which $a$ is active. It follows that for every directed path in $T_i$, the union of the lifetimes of the corresponding jobs is a single interval (i.e., it is connected, with no gaps). The same is true for the union of all of the lifetimes of jobs in $T_i$ (it is true for each root-leaf path, and hence for their union).

**Step 2.** Let $[s_i, t]$ denote the union of lifetimes of all jobs in $T_i$. Note that $s_i$ is simply the earliest release date of a job in $T_i$. We claim that, conversely, every job processed by the Balance Algorithm in the interval $[s_i, t]$ appears in $T_i$. This follows from an easy proof by contradiction. If $j \notin T_i$ and is processed at time $s \in [s_i, t]$, then $s$ is pre-empting some job of $T_i$ — by the definition of $[s_i, t]$, $s$ is during the lifetime of some job of $T_i$. But then $j \in T_i$ by the definition of an exchange tree.

**Step 3.** In this step we make use of the fact that the Balance Algorithm has a faster machine than SRPT. It is essentially a counting argument.

The Balance Algorithm never idles when there is an active job. Steps 1 and 2 imply that the Balance Algorithm works continually throughout the interval $[s_i, t]$, and solely on jobs of $T_i$. A total of $(1 + \epsilon)(t - s_i)$ units of work are completed on these jobs. Let $W_1$ and $W_2$ denote the amount of this work devoted to jobs of $(X \setminus Y) \cap T_i$ and of $(X \cup Y)^c \cap T_i$, respectively.

Now consider the work accomplished by SRPT during $[s_i, t]$. Since it completes all jobs of $T_i$ outside of $Y$ by time $t$, it spends at least as much on these jobs as does the Balance Algorithm during $[s_i, t]$ (namely, $W_1 + W_2$). Since SRPT only has a unit-speed machine, the total amount of work it spends on any jobs in the interval $[s_i, t]$ is at most $t - s_i$. Thus, the Balance Algorithm must spend at least

$$[(1 + \epsilon)(t - s_i) - W_1 - W_2] - [(t - s_i) - W_1 - W - 2] = \epsilon(t - s_i)$$

more work on jobs of $Y \cap T_i$ than does SRPT in the interval $[s_i, t]$. Using $w(j)$ to denote the weight of job $j$ at time $t$, we have

$$\begin{aligned}\sum_{j \in Y \cap T_i} w(j) &\geq \epsilon(t - s_i) \\ &\geq \epsilon \cdot W_1 \\ &= \epsilon \sum_{j \in (X \setminus Y) \cap T_i} w(j) \\ &\geq \sum_{j=1}^{i} w(j), \end{aligned} \tag{8}$$

where in the final inequality we use the fact that all of the jobs $\{1, 2, \ldots, i-1\}$ appear in (level 1 of) the exchange tree $T_i$. (Proof: by (7), every job of $\{1, 2, \ldots, i-1\}$ appears at the same time or later than job $i$, and the Balance Algorithm always immediately switches to a job that just arrived.)

Now, inequality (8) holds for every job $i = 1, 2, \ldots, k$ of $X \setminus Y$. This means that, inductively for each job $i = 1, 2, \ldots, k$, we can "charge" $\epsilon w(i)$ units of work to the work done by the Balance Algorithm on the jobs of $Y \cap T_i$ in $[s, t_i]$. Formally, that (8) holds for every $i$ means that there exist nonnegative "charges" $g(i, j)$ satisfying the following properties:

(P1) $\sum_{j \in Y} g(i, j) = \epsilon w(j)$ for every job $i = 1, 2, \ldots, k$;

(P2) $\sum_{i=1}^{k} g(i, j) \leq w(j)$ for every job $j \in Y$; and

(P3) $g(i, j) > 0$ only if $j \in Y \cap T_i$.

**Step 4.** The fourth step uses properties of the Balance Algorithm. The key claim is: for every job $i = 1, 2, \ldots, k$ of $X \setminus Y$ and for every job $j \in T_i$ in the corresponding exchange tree, the weight $w(i)$ of $i$ at time $t$ is at least that $w(j)$ of $j$ at time $t$.

First, for intuition, consider a job $j$ at level 1 of $T_i$. This means that at some point $s$ during $i$'s lifetime, job $j$ was processed by the Balance Algorithm instead of job $i$. By the definition of the algorithm (Figure 1), this means that $j$'s weight was at most that of $i$ at time $s$. The Balance Algorithm maintains this inequality as an invariant at least until job $i$ completes (if $j$'s weight catches up with that of $i$, they are processed at the same rate). Since $i \notin X$ it cannot complete before time $t$, so $w(j) \leq w(i)$ (at time $t$).

The general argument is by induction on the level $\ell$ of the exchange tree $T_i$. The base case of $\ell = 0$ is trivial. For the inductive step, consider a job $j_\ell$ in the $\ell$th level of $T_i$, with $i, j_1, \ldots, j_{\ell-1}, j_\ell$ denoting the jobs on the $i$-$j_\ell$ path of $T_i$. The inductive hypothesis implies that, for all $h \leq \ell - 1$ and $s \leq t$, the weight of $j_h$ at time $s$ is at most $w(i)$, the weight of $i$ at time $t$. Since the edge $j_{\ell-1} \rightarrow j_\ell$ appears in $T_i$, $j_\ell$ is processed at some point during the lifetime of $j_{\ell-1}$; let $s^*$ be the earliest such time. By the definition of the Balance Algorithm, the weight of $j_\ell$ is at most that of $j_{\ell-1}$ at time $s^*$. By Step 1, the union of the lifetimes of $i, j_1, \ldots, j_{\ell-1}$ is an interval that includes $[s^*, t]$ (since $j_{\ell-1}$ is active at time $s^*$). Thus whenever $j_\ell$ is processed during this interval, its weight is bounded above by that of some job in $\{i, j_1, \ldots, j_{\ell-1}\}$ at that time. Thus $j_\ell$'s weight remains bounded above by $w(i)$ up until time $t$.

**Proof of Theorem 4.1.** The proof of (6), and hence of Theorem 4.1, follows easily from Steps 3 and 4. We can write

$$\begin{aligned}
|X \setminus Y| = k &= \sum_{i=1}^{k} \frac{\epsilon w(i)}{\epsilon w(i)} \\
&= \frac{1}{\epsilon} \sum_{i=1}^{k} \sum_{j \in Y \cap T_i} \frac{g(i,j)}{w(i)} \qquad (9) \\
&\leq \frac{1}{\epsilon} \sum_{i=1}^{k} \sum_{j \in Y \cap T_i} \frac{g(i,j)}{w(j)} \qquad (10) \\
&\leq \frac{1}{\epsilon} \sum_{j \in Y} \frac{1}{w(j)} \underbrace{\sum_{i=1}^{k} g(i,j)}_{\leq w(j) \text{ by (P2)}} \\
&\leq \frac{|Y|}{\epsilon},
\end{aligned}$$

where (9) follows from properties (P1) and (P3) of the $g(i,j)$'s and (10) follows from the fact that all jobs of $T_i$ have weight at most $w(i)$ (Step 4). Thus

$$|X| \leq |X \setminus Y| + |Y| \leq \left(1 + \frac{1}{\epsilon}\right) \cdot |Y|,$$

which completes the proof.