Beyond the Worst-Case Analysis of Algorithms

Edited by
Tim Roughgarden

Contents

1
Resource Augmentation

Tim Roughgarden

Abstract

This chapter introduces *resource augmentation*, where the performance of an algorithm is compared to the best-possible solution that is handicapped by less resources. We consider three case studies: online paging, with cache size as the resource; selfish routing, with capacity as the resource; and scheduling, with processor speed as the resource. Resource augmentation bounds also imply "loosely competitive" bounds, which show that an algorithm's performance is near-optimal for most resource levels.

1.1 Online Paging Revisited

This section illustrates the idea of resource augmentation with a familiar example, the competitive analysis of online paging algorithms. Section 1.2 discusses the pros and cons of resource augmentation more generally, Sections 1.3 and 1.4 describe additional case studies in routing and scheduling, and Section 1.5 shows how resource augmentation bounds lead to "loosely competitive" guarantees.

1.1.1 The Model

Our first case study of resource augmentation concerns the online paging problem introduced in Chapter 1. Recall the ingredients of the problem:

- There is a slow memory with N pages.
- There is a fast memory (a *cache*) that can hold only $k < N$ of the pages at a time.
- Page requests arrive online over time, with one request per time step. The decisions of an online algorithm at time t can depend only on the requests arriving at or before time t.
- If the page p_t requested at time t is already in the cache, no action is necessary.

- If p_t is not in the cache, it must be brought in; if the cache is full, one of its k pages must be evicted. This is called a *page fault.*[1]

We measure the performance $\textsc{Perf}(A, z)$ of an algorithm A on a page request sequence z by the number of page faults incurred.

1.1.2 FIF and LRU

As a benchmark, what would we do if we had clairvoyance about all future page requests? An intuitive greedy algorithm minimizes the number of page faults.

Theorem 1.1 (Bélády (1967)) *The* Furthest-in-the-Future (FIF) *algorithm, which on a page fault evicts the page to be requested furthest in the future, always minimizes the number of page faults.*

The FIF algorithm is not an online algorithm, as its eviction decisions depend on future page requests. The *Least Recently Used (LRU)* policy, which on a page fault evicts the page whose most recent request is furthest in the past, is an online surrogate for the FIF algorithm that uses the past as an approximation for the future. Empirically, the LRU algorithm performs well on most "real-world" page request sequences—not much worse than the unimplementable FIF algorithm, and better than other online algorithms such as first-in first-out (FIFO). The usual explanation for the superiority of the LRU algorithm is that the page request sequences that arise in practice exhibit locality of reference, with recent requests likely to be requested again soon, and that LRU automatically adapts to and exploits this locality.

1.1.3 Competitive Ratio

One popular way to assess the performance of an online algorithm is through its competitive ratio:[2]

Definition 1.2 (Sleator and Tarjan (1985)) The *competitive ratio* of an online algorithm A is its worst-case performance (over inputs z) relative to an optimal *offline* algorithm OPT that has advance knowledge of the entire input:

$$\max_z \frac{\textsc{Perf}(A, z)}{\textsc{Perf}(OPT, z)}.$$

[1] This model corresponds to "demand paging," meaning algorithms that modify the cache only in response to a page fault. The results in this section continue to hold in the more general model in which an algorithm is allowed to make arbitrary changes to the cache at each time step, whether or not there is a page fault, with the cost incurred by the algorithm equal to the number of changes.

[2] See Chapter 24 for a deep dive on alternatives to worst-case analysis in the competitive analysis of online algorithms.

For the objective of minimizing the number of page faults, the competitive ratio is always at least 1, and the closer to 1 the better.[3]

Exercise 1.1 of Chapter 1 shows that, for every deterministic online paging algorithm A and cache size k, there are arbitrarily long page request sequences z such that A faults at every time step while the FIF algorithm faults at most once per k time steps. This example shows that every deterministic online paging algorithm has a competitive ratio of at least k. For most natural online algorithms, there is a matching upper bound of k. This state of affairs is unsatisfying for several reasons:

1. The analysis gives an absurdly pessimistic performance prediction for LRU (and all other deterministic online algorithms), suggesting that a 100% page fault rate is unavoidable.
2. The analysis suggests that online algorithms perform worse (relative to FIF) as the cache size grows, a sharp departure from empirical observations.
3. The analysis fails to differentiate between competing policies like LRU and FIFO, which both have a competitive ratio of k.

We next address the first two issues through a resource augmentation analysis (but not the third, see Exercise 1.2).

1.1.4 A Resource Augmentation Bound

In a *resource augmentation* analysis, the idea is to compare the performance of a protagonist algorithm (like LRU) to an all-knowing optimal algorithm that is handicapped by “less resources.” Naturally, weakening the capabilities of the offline optimal algorithm can only lead to better approximation guarantees.

Let $\textsc{Perf}(A, k, z)$ denote the number of page faults incurred by the algorithm A with cache size k on the page request sequence z. The main result of this section is:

Theorem 1.3 (Sleator and Tarjan (1985)) *For every page request sequence z and cache sizes $h \le k$,*

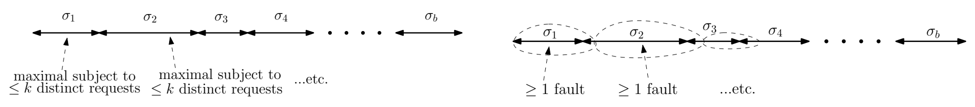
$$\textsc{Perf}(LRU, k, z) \le \frac{k}{k-h+1} \cdot \textsc{Perf}(FIF, h, z),$$

plus an additive error term that goes to 0 with $\textsc{Perf}(FIF, h, z)$.

For example, LRU suffers at most twice as many page faults as the unimplementable FIF algorithm when the latter has roughly half the cache size.

Proof Consider an arbitrary page request sequence z and cache sizes $h \le k$. We first prove an upper bound on the number of page faults incurred by the LRU algorithm, and then a lower bound on the number of faults incurred by the FIF

[3] One usually ignores any extra additive terms in the competitive ratio, which vanish as $\textsc{Perf}(OPT, z) \to \infty$.



(a) Blocks of a request sequence (b) Lower bound for FIF (with $h = k$)

Figure 1.1 Proof of Theorem 1.3. In (a), the blocks of a page request sequence; the LRU algorithm incurs at most k page faults in each. In (b), the FIF algorithm incurs at least $k - h + 1$ page faults in each "shifted block."

algorithm. A useful idea for accomplishing both goals is to break z into *blocks* $\sigma_1, \sigma_2, \ldots, \sigma_b$. Here σ_1 is the maximal prefix of z in which only k distinct pages are requested; the block σ_2 starts immediately after and is maximal subject to only k distinct pages being requested (ignoring what was requested in σ_1); and so on.

For the first step, note that LRU faults at most k times within a single block—at most once per page requested in the block. The reason is that once a page is brought into the cache, LRU won't evict it until k other distinct pages are requested, and this can't happen until the following block. Thus LRU incurs at most bk page faults, where b is the number of blocks. See Figure 1.1(a).

For the second step, consider the FIF algorithm with a cache size $h \leq k$. Consider the first block σ_1 plus the first request of the second block σ_2. Since σ_1 is maximal, this represents requests for $k + 1$ distinct pages. At least $k - h + 1$ of these pages are initially absent from the size-h cache, so no algorithm can serve all $k+1$ pages without incurring at least $k - h + 1$ page faults. Similarly, suppose the first request of σ_2 is the page p. After an algorithm serves the request for p, the cache contains only $h - 1$ pages other than p. By the maximality of σ_2, the "shifted block" comprising the rest of σ_2 and the first request of σ_3 includes requests for k distinct pages other than p; these cannot all be served without incurring another

$$\underbrace{k}_{\text{requests other than } p} \quad - \quad \underbrace{(h-1)}_{\text{pages in cache other than } p}$$

page faults. And so on, resulting in at least $(b - 1)(k - h + 1)$ page faults overall. See Figure 1.1(b).

We conclude that

$$\textsc{Perf}(LRU, k, z) \leq bk \leq \frac{k}{k-h+1} \cdot \textsc{Perf}(FIF, h, z) + \frac{k}{(b-1)(k-h+1)}.$$

The additive error term goes to 0 with b, and the proof is complete. $\square$

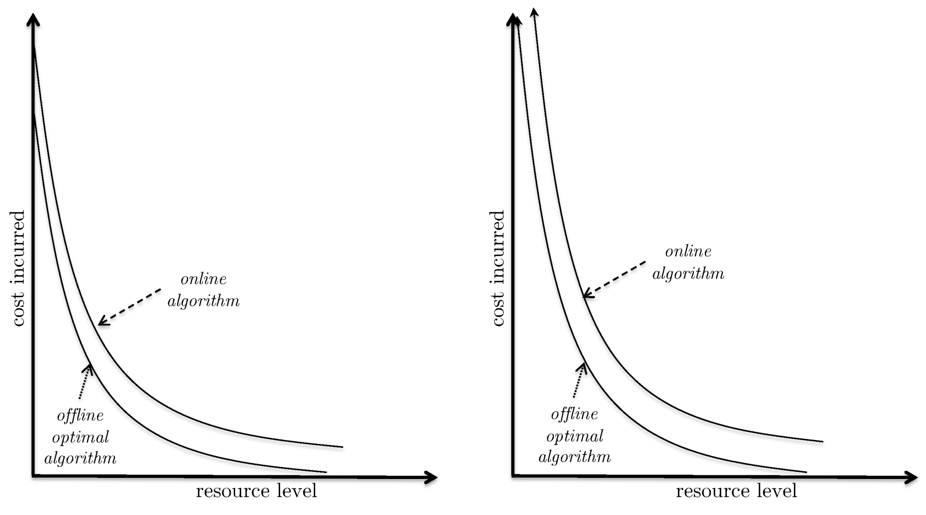


Figure 1.2 Competitive ratio guarantees vs. resource augmentation guarantees. All curves plot, for a fixed input, the cost incurred by an algorithm (e.g., number of page faults) as a function of the resource level (e.g., the cache size). In (a), a good upper bound on the competitive ratio requires that the curve for the online algorithm closely approximates that of the offline optimal algorithm pointwise over the x-axis. In (b), the vertical distance between the two curves (and hence the competitive ratio) grows large as the resource level approaches its minimum. A resource augmentation guarantee roughly translates to the relaxed requirement that every point of the online algorithm's performance curve has a nearby neighbor somewhere on the optimal offline algorithm's performance curve.

1.2 Discussion

Resource augmentation guarantees make sense for any problem where there's a natural notion of a "resource," with algorithm performance improving in the resource level; see Sections 1.3 and 1.4 for two further examples. In general, a resource augmentation guarantee implies that the performance curves (i.e., performance as a function of resource level) of an online algorithm and the offline optimal algorithm are similar (Figure 1.2).

The resource augmentation guarantees in this chapter resemble worst-case analysis, in that no model of data is proposed; the difference is purely in the method of measuring algorithm performance (relative to optimal performance). As usual, this is both a feature and a bug: the lack of a data model guarantees universal applicability, but also robs the analyst of any opportunity to articulate properties of "real-world" inputs that might lead to a more accurate and fine-grained analysis. There is nothing inherently worst-case about resource augmentation guarantees, however, and the concept can equally well be applied with one of the models of data discussed in the other parts of this book.[4]

How should you interpret a resource augmentation guarantee like Theorem 1.3? Should you be impressed? Taken at face value, Theorem 1.3 seems much more meaningful than the competitive ratio of k without resource augmentation, even though it doesn't provide particularly sharp performance predictions (as to be expected, given the lack of a model of data). But isn't it an "apples vs. oranges" comparison? The optimal offline algorithm is powerful in its knowledge of all future page requests, but it's artificially hobbled by a small cache.

One interpretation of a resource augmentation guarantee is as a two-step recipe for building a system in which an online algorithm has good performance.

1. Estimate the resource level (e.g., cache size) such that the optimal offline algorithm has acceptable performance (e.g., page fault rate below a given target).[5] This task can be simpler than reasoning simultaneously about the cache size and paging algorithm design decisions.
2. Scale up the resources to realize the resource augmentation guarantee (e.g., doubling the cache size needed by the FIF algorithm to achieve good performance).

A second justification for resource augmentation guarantees is that they usually lead directly to good "apples vs. apples" comparisons for most resource levels (as suggested by Figure 1.2(b)). Section 1.5 presents a detailed case study in the context of online paging.

1.3 Selfish Routing

Our second case study of a resource augmentation guarantee concerns a model of *selfish routing* in a congested network.

1.3.1 The Model and a Motivating Example

In selfish routing, we consider a directed flow network $G = (V, E)$, with r units of flow traveling from a source vertex s to a sink vertex t; r is called the *traffic rate.* Each edge e of the network has a flow-dependent cost function $c_e(x)$. For example, in the network in Figure 1.3(a), the top edge has a constant cost function $c(x) = 1$, while the cost to traffic on the bottom edge equals the amount of flow x on the edge.

The key approximation concept in selfish routing networks is the *price of anarchy*

[4] For example, Chapter 27 combines robust distributional analysis with resource augmentation, in the context of prior-independent auctions.
[5] Remember: competing with the optimal algorithm is only useful when its performance is good in some absolute sense!

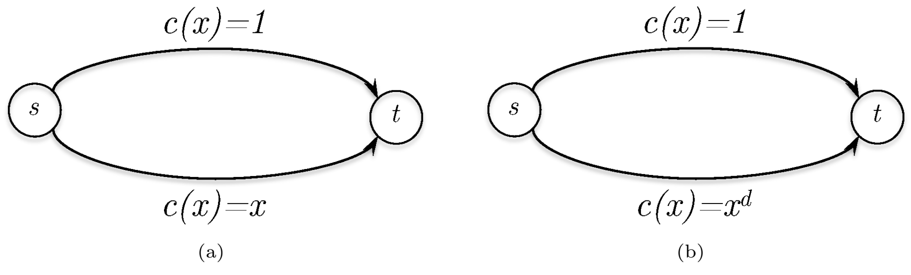


Figure 1.3 Two selfish routing networks. Each cost function $c(x)$ describes the cost incurred by users of an edge, as a function of the amount of traffic routed on that edge.

which, as usual with approximation ratios, is defined as the ratio between two things: a realizable protagonist and a hypothetical benchmark.

Our protagonist is an *equilibrium flow*, in which all traffic is routed on shortest paths, where the length of an s-t path P is the (flow-dependent) quantity $\sum_{e \in P} c_e(f_e)$, where f_e denotes the amount of flow using the edge e. In Figure 1.3(a), with one unit of traffic, the only equilibrium flow sends all traffic on the bottom edge. If $\epsilon > 0$ units of traffic were routed on the top path, that traffic would not be routed on a shortest path (incurring cost 1 instead of $1 - \epsilon$), and hence would want to switch paths.

Our benchmark is the optimal solution, meaning the fractional s-t flow that routes the r units of traffic to minimize the total cost $\sum_{e \in E} c_e(f_e) f_e$. For example, in Figure 1.3(a), the optimal flow splits traffic evenly between the two paths, for a cost of $\frac{1}{2} \cdot 1 + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$. The cost of the equilibrium flow is $0 \cdot 1 + 1 \cdot 1 = 1$.

The price of anarchy of a selfish routing network is defined as the ratio between the cost of an equilibrium flow and that of an optimal flow.[6] In the network in Figure 1.3(a), the price of anarchy is 4/3.

An interesting research goal is to identify selfish routing networks in which the price of anarchy is close to 1—networks in which decentralized optimization by selfish users performs almost as well as centralized optimization. Unfortunately, without any restrictions on edges' cost functions, the price of anarchy can be arbitrarily large. To see this, replace the cost function on the bottom edge in Figure 1.3(a) by the function $c(x) = x^d$ for a large positive integer d (Figure 1.3(b)). The equilibrium flow and its cost remain the same, with all selfish traffic using the bottom edge for an overall cost of 1. The optimal flow, however, improves with d: Routing $1 - \epsilon$ units of flow on the bottom edge and ϵ units on the top edge yields a flow with cost $\epsilon + (1 - \epsilon)^{d+1}$. This cost tends to 0 as d tends to infinity and ϵ tends appropriately to 0, and hence the price of anarchy goes to infinity with d.

[6] It turns out that the equilibrium flow cost is uniquely defined in every selfish routing network with continuous and nondecreasing edge cost functions; see the Notes for details.

1.3.2 A Resource Augmentation Guarantee

Despite the negative example above, a very general resource augmentation guarantee holds in selfish routing networks.[7]

Theorem 1.4 (Roughgarden and Tardos (2002)) *For every network G with nonnegative, continuous, and nondecreasing cost functions, for every traffic rate $r > 0$, and for every $\delta > 0$, the cost of an equilibrium flow in G with traffic rate r is at most $\frac{1}{\delta}$ times the cost of an optimal flow with traffic rate $(1 + \delta)r$.*

For example, consider the network in Figure 1.3(b) with $r = \delta = 1$ (and large d). The cost of the equilibrium flow with traffic rate 1 is 1. The optimal flow can route one unit of traffic cheaply (as we've seen), but then the network gets 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 with double the traffic exceeds that of the original equilibrium flow.

Theorem 1.4 can be reformulated as a comparison between an equilibrium flow in a network with "faster" edges and an optimal flow in the original network. For example, simple calculations (Exercise 1.5) show that the following statement is equivalent to Theorem 1.4 with $\delta = 1$.

Corollary 1.5 *For every network G with nonnegative, continuous, and nondecreasing cost functions and for every traffic rate $r > 0$, the cost of an equilibrium flow in G with traffic rate r and cost functions $\{\tilde{c}_e\}_{e\in E}$ is at most that of an optimal flow in G with traffic rate r and cost functions $\{c_e\}_{e\in E}$, where each function $\tilde{c}_e$ is derived from c_e as $\tilde{c}_e(x) = c_e(x/2)/2$.*

Corollary 1.5 takes on a particularly appealing form in networks with M/M/1 delay functions, meaning cost functions of the form $c_e(x) = 1/(u_e - x)$, where u_e can be interpreted as an edge capacity or a queue service rate. (If $x \geq u_e$, interpret $c_e(x)$ as $+\infty$.) In this case, the modified function $\tilde{c}_e$ in Corollary 1.5 is

$$\tilde{c}_e(x) = \frac{1}{2(u_e - \frac{x}{2})} = \frac{1}{2u_e - x}.$$

Corollary 1.5 thus translates to the following design principle for selfish routing networks with M/M/1 delay functions: to outperform optimal routing, double the capacity of every edge.

1.3.3 Proof of Theorem 1.4 (Parallel Edges)

As a warm-up to the proof of Theorem 1.4, consider the special case where $G = (V, E)$ is a network of parallel edges, meaning $V = \{s, t\}$ and every edge of E is directed from s to t (as in Figure 1.3). Choose a traffic rate $r > 0$; a cost function c_e

[7] This result holds still more generally, in networks with multiple source and sink vertices (Exercise 1.4).

for each edge $e \in E$ that is nonnegative, continuous, and nondecreasing; and the parameter $\delta > 0$. Let f and f^* denote equilibrium and optimal flows in G at traffic rates r and $(1+\delta)r$, respectively. The equilibrium flow f routes traffic only on shortest paths, so there is a number L (the shortest s-t path length) such that

$$\begin{aligned} c_e(f_e) &= L \quad \text{if } f_e > 0; \\ c_e(f_e) &\geq L \quad \text{if } f_e = 0. \end{aligned}$$

The cost of the equilibrium flow f is then

$$\sum_{e \in E} c_e(f_e) f_e = \sum_{e \in E \,:\, f_e > 0} c_e(f_e) f_e = \sum_{e \in E \,:\, f_e > 0} L \cdot f_e = r \cdot L,$$

as the total amount of flow $\sum_{e \,:\, f_e > 0} f_e$ equals the traffic rate r.

How can we bound from below the cost of the optimal flow f^*, relative to the cost rL of f? To proceed, bucket the edges of E into two categories:

$$\begin{aligned} E_1 \quad &:= \text{the edges } e \text{ with } f_e^* \geq f_e; \\ E_2 \quad &:= \text{the edges } e \text{ with } f_e^* < f_e. \end{aligned}$$

With so few assumptions on the network cost functions, we can't say much about the costs of edges under the optimal flow f^*. The two things we *can* say are that $c_e(f_e^*) \geq L$ for all $e \in E_1$ (because cost functions are nondecreasing) and that $c_e(f_e^*) \geq 0$ for all $e \in E_2$ (because cost functions are nonnegative). At the very least, we can therefore lower bound the cost of f^* by

$$\sum_{e \in E} c_e(f_e^*) f_e^* \geq \sum_{e \in E_1} c_e(f_e^*) f_e^* \geq L \cdot \sum_{e \in E_1} f_e^*. \tag{1.1}$$

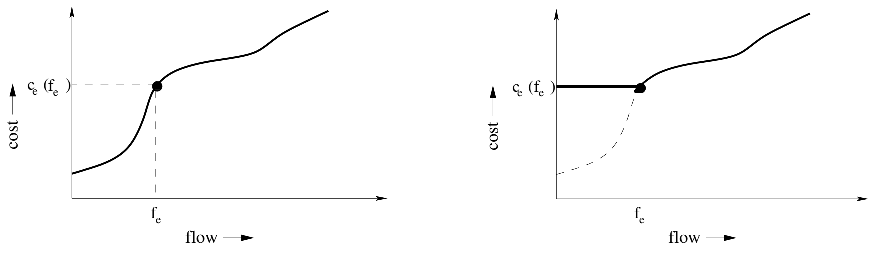
How little traffic could f^* possibly route on the edges of E_1? The flow routes $(1+\delta)r$ units of traffic overall. It routes less flow than f on the edges of E_2 (by the definition of E_2), and f routes at most r units (i.e., its full traffic rate) on these edges. Thus

$$\sum_{e \in E_1} f_e^* = (1+\delta)r - \sum_{e \in E_2} f_e^* \geq (1+\delta)r - \underbrace{\sum_{e \in E_2} f_e}_{\leq r} \geq \delta r. \tag{1.2}$$

Combining the inequalities (1.1) and (1.2) shows that the cost of f^* is at least $\delta \cdot rL$, which is δ times the cost of f, as desired.

1.3.4 Proof of Theorem 1.4 (General Networks)

Consider now the general case of Theorem 1.4, in which the network $G = (V, E)$ is arbitrary. General networks are more complex than networks of parallel edges because there is no longer a one-to-one correspondence between edges and paths—a path might comprise many edges, and an edge might participate in many different paths. This complication aside, the proof proceeds similarly to that for the special case of networks of parallel edges.



(a) Graph of cost function c_e and its value at flow value f_e

(b) Graph of cost function $\bar{c}_e$

Figure 1.4 Construction in the proof of Theorem 1.4 of the fictitious cost function $\bar{c}_e$ from the original cost function c_e and equilibrium flow value f_e.

Fix a traffic rate r, a cost function c_e for each edge $e \in E$, and the parameter $\delta > 0$. As before, let f and f^* denote equilibrium and optimal flows in G at traffic rates r and $(1+\delta)r$, respectively. It is still true that there is a number L such that all traffic in f is routed on paths P with length $\sum_{e \in P} c_e(f_e)$ equal to L, and such that all s-t paths have length at least L. The cost of the equilibrium flow is again rL.

The key trick in the proof is to replace, for the sake of analysis, each cost function $c_e(x)$ (Figure 1.4(a)) by the larger cost function $\bar{c}_e(x) = \max\{c_e(x), c_e(f_e)\}$ (Figure 1.4(b)). This trick substitutes for the decomposition in Section 1.3.3 of E into E_1 and E_2. With the fictitious cost functions $\bar{c}_e$, edge costs are always as large as if the equilibrium flow f had already been routed in the network.

By design, the cost of the optimal flow f^* is easy to bound from below with the fictitious cost functions. Even with zero flow in the network, every s-t path has cost at least L with respect to these functions. Because f^* routes $(1+\delta)r$ units of traffic on paths with (fictitious) cost at least L, its total (fictitious) cost with respect to the $\bar{c}_e$'s is at least $(1+\delta)rL$.

We can complete the proof by showing that the fictitious cost of f^* (with respect to the $\bar{c}_e$'s) exceeds its real cost (with respect to the c_e's) by at most rL, the equilibrium flow cost. For each edge $e \in E$ and $x \geq 0$, $\bar{c}_e(x) - c_e(x)$ is either 0 (if $x \geq f_e$) or bounded above by $c_e(f_e)$ (if $x < f_e$); in any case,

$$\underbrace{\bar{c}_e(f_e^*)f_e^*}_{\text{fictitious cost of } f^* \text{ on } e} \quad - \quad \underbrace{c_e(f_e^*)f_e^*}_{\text{real cost of } f^* \text{ on } e} \quad \leq \quad \underbrace{c_e(f_e)f_e}_{\text{real cost of } f \text{ on } e} \quad .$$

Summing this inequality over all edges $e \in E$ shows that the difference between the costs of f^* with respect to the different cost functions is at most the cost of f (i.e., rL); this completes the proof of Theorem 1.4.

1.4 Speed Scaling in Scheduling

The lion's share of killer applications of resource augmentation concern scheduling problems. This section describes one paradigmatic example.

1.4.1 Non-Clairvoyant Scheduling

We consider a model with a single machine and m jobs that arrive online. Each job j has a release time r_j and the algorithm is unaware of the job before this time. Each job j has a processing time p_j, indicating how much machine time is necessary to complete it. 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 basic objective of minimizing the total flow time:[8]

$$\sum_{j=1}^{m} (C_j - r_j)\,,$$

where C_j denotes the completion time of job j. For an alternative formulation, note that each infinitesimal time interval $[t, t+dt]$ contributes dt to the flow time $C_j - r_j$ of every job that is *active* at time t, meaning released but not yet completed. Thus, the total flow time can be written as

$$\int_0^\infty |X_t| dt, \tag{1.3}$$

where X_t denotes the active jobs at time t.

The *shortest remaining processing time (SRPT)* algorithm always processes the job that is closest to completion (preempting jobs as needed). This algorithm makes $|X_t|$ as small as possible for all times t (Exercise 1.7) and is therefore optimal. This is a rare example of a problem where the optimal offline algorithm is implementable as an online algorithm.

SRPT uses knowledge of the job processing times to make decisions, and as such is a *clairvoyant* algorithm. What about applications in which a job's processing time is not known before it completes, where a *non-clairvoyant* algorithm is called for? No non-clairvoyant online algorithm can guarantee a total flow time close to that achieved by SRPT (Exercise 1.8). Could a resource augmentation approach provide more helpful algorithmic guidance?

1.4.2 A Resource Augmentation Guarantee for SETF

The natural notion of a "resource" in this scheduling problem is processor speed. Thus, a resource augmentation guarantee would assert that the total flow time of

[8] This objective is also called the total response time.

some non-clairvoyant protagonist *with a faster machine* is close to that of SRPT with the original machine.

We prove such a guarantee for the *shortest elapsed time first (SETF)* algorithm, which always processes the job that has been processed the least so far. When multiple jobs are tied for the minimum elapsed time, the machine splits its processing power equally between them. SETF does not use jobs' processing times to make decisions, and as such is a non-clairvoyant algorithm.

Example 1.6 Fix parameters $\epsilon, \delta > 0$, with δ much smaller than ϵ. With an eye toward a resource augmentation guarantee, we compare the total flow time of SETF with a machine with speed $1+\epsilon$—meaning that the machine can process $(1+\epsilon)t$ units of jobs in a time interval of length t—to that of SRPT with a unit-speed machine.

Suppose m jobs arrive at times $r_1 = 0, r_2 = 1, \ldots, r_m = m-1$, where m is $\lfloor \frac{1}{\epsilon} \rfloor - 1$. Suppose $p_j = 1+\epsilon+\delta$ for every job j. Under the SRPT algorithm, assuming that $\epsilon + \delta$ is sufficiently small, there will be at most 2 active jobs at all times (the most recently released jobs); using (1.3), the total flow time of its schedule is $O(\frac{1}{\epsilon})$. The SETF algorithm will not complete any jobs until after time m, so in each time interval $[j-1, j]$ there are j active jobs. Using (1.3) again, the total flow time of SETF's schedule is $\Omega(\frac{1}{\epsilon^2})$.

Example 1.6 shows that SETF is not optimal, and it draws a line in the sand: The best we can hope for is that the SETF algorithm with a $(1+\epsilon)$-speed machine achieves total flow time $O(\frac{1}{\epsilon})$ times that suffered by the SRPT algorithm with a unit-speed machine. The main result of this section states that this is indeed the case.

Theorem 1.7 (Kalyanasundaram and Pruhs (2000)) *For every input and $\epsilon > 0$, the total flow time of the schedule produced by the SETF algorithm with a machine with speed $1+\epsilon$ is at most*

$$1 + \frac{1}{\epsilon}$$

times that by the SRPT algorithm with a unit-speed machine.

Using the second version (1.3) of the objective function, Theorem 1.7 reduces to the following pointwise (over time) bound.

Lemma 1.8 *Fix $\epsilon > 0$. For every input, at every time step t,*

$$|X_t| \leq \left(1 + \frac{1}{\epsilon}\right) |X_t^*|,$$

where X_t and X_t^* denote the jobs active at time t under SETF with a $(1+\epsilon)$-speed machine and SRPT with a unit-speed machine, respectively.

In Example 1.6, at time $t = m$, $|X_t^*| = 1$ (provided ϵ, δ are sufficiently small) while $|X_t| = m \approx \frac{1}{\epsilon}$. Thus, every inequality used in the proof of Lemma 1.8 should hold almost with equality for the instance in Example 1.6. The reader is encouraged to keep this example in mind throughout the proof.

To describe the intuition behind Lemma 1.8, fix a time t. Roughly:

1. SRPT must have spent more time processing the jobs of $X_t \setminus X_t^*$ than SETF (because SRPT finished them by time t while SETF did not).
2. SETF performed $1 + \epsilon$ times as much job processing as SRPT, an ϵ portion of which must have been devoted to the jobs of X_t^*.
3. Because SETF prioritizes the jobs that have been processed the least, it also spent significant time processing the jobs of $X_t \setminus X_t^*$.
4. SRPT had enough time to complete all the jobs of $X_t \setminus X_t^*$ by time t, so there can't be too many such jobs.

The rest of this section supplies the appropriate details.

1.4.3 Proof of Lemma 1.8: Preliminaries

Fix an input and a time t, with X_t and X_t^* defined as in Lemma 1.8. Rename the jobs of $X_t \setminus X_t^* = \{1, 2, \ldots, k\}$ such that $r_1 \geq r_2 \geq \cdots \geq r_k$.

Consider the execution of the SETF algorithm with a $(1 + \epsilon)$-speed machine. We say that job ℓ *interferes* with job j if there is a time $s \leq t$ at which j is active and ℓ is processed in parallel with or instead of j. The *interference set* I_j of a job j is the transitive closure of the interference relation:

1. Initialize I_j to $\{j\}$.
2. While there is a job ℓ that interferes with a job of I_j, add one such job to I_j.

In Example 1.6 with $t = +\infty$, the interference set of every job is the set of all jobs (because all of the jobs are processed in parallel at the very end of the algorithm). If instead $t = m$, then $I_j = \{j, j + 1, \ldots, m\}$ for each job $j \in \{1, 2, \ldots, m\}$.

The interference set of a job is uniquely defined, independent of which interfering job is chosen in each iteration of the while loop. Note that the interference set can contain jobs that were completed by SETF strictly before time t.

We require several properties of the interference sets of the jobs in $X_t \setminus X_t^*$. To state the first, define the *lifetime* of a job j as the interval $[r_j, \min\{C_j, t\}]$ up to time t during which it is active.

Proposition 1.9 *Let $j \in \{1, 2, \ldots, k\}$ be a job of $X_t \setminus X_t^*$. The union of the lifetimes of the jobs in an interference set I_j is the interval $[s_j, t]$, where s_j is the earliest release time of a job in I_j.*

Proof One job can interfere with another only if their lifetimes overlap. By induction, the union of the lifetimes of jobs in I_j is an interval. The right endpoint of the interval is at most t by definition, and is at least t because job j is active at time t. The left endpoint of the interval is the earliest time at which a job of I_j is active, which is $\min_{\ell \in I_j} r_\ell$. □

Conversely, every job processed in the interval corresponding to an interference set belongs to that set.

Proposition 1.10 *Let $j \in \{1, 2, \ldots, k\}$ be a job of $X_t \setminus X_t^*$ and $[s_j, t]$ the union of the lifetimes of the jobs in j's interference set I_j. Every job processed at some time $s \in [s_j, t]$ belongs to I_j.*

Proof Suppose job ℓ is processed at some time $s \in [s_j, t]$. Since $[s_j, t]$ is the union of the lifetimes of the jobs in I_j, I_j contains a job i that is active at time s. If $i \neq \ell$, then job ℓ interferes with i and hence also belongs to I_j. □

The next proposition helps implement the third step of the intuition outlined in Section 1.4.2.

Proposition 1.11 *Let $j \in \{1, 2, \ldots, k\}$ be a job of $X_t \setminus X_t^*$. Let w_ℓ denote the elapsed time of a job ℓ under SETF by time t. Then $w_\ell \leq w_j$ for every job ℓ in j's interference set I_j.*

Proof We proceed by induction on the additions to the interference set. Consider an iteration of the construction that adds a job j_1 to I_j. By construction, there is a sequence of already-added jobs $j_2, j_3, \ldots, j_p$ such that $j_p = j$ and j_i interferes with j_{i+1} for each $i = 1, 2, \ldots, p-1$. (Assume that $p > 1$; otherwise we're in the base case where $j_1 = j$ and there's nothing to prove.) As in Proposition 1.9, the union of the lifetimes of the jobs $\{j_2, j_3, \ldots, j_p\}$ forms an interval $[s, t]$; the right endpoint is t because $j_p = j$ is active at time t. By induction, $w_{j_i} \leq w_j$ for every $i = 2, 3, \ldots, p$. Thus, whenever j_1 is processed in the interval $[s, t]$, there is an active job with elapsed time at most w_j. By virtue of being processed by SETF, the elapsed time of j_1 at any such point in time is also at most w_j. The job j_1 must be processed at least once during the interval $[s, t]$ (as the job interferes with j_2), so its elapsed time by time t is at most w_j. □

1.4.4 Proof of Lemma 1.8: The Main Argument

We are now prepared to implement formally the intuition outlined in Section 1.4.2.

Fix a job $j \in X_t \setminus X_t^*$; recall that $X_t \setminus X_t^* = \{1, 2, \ldots, k\}$, with jobs indexed in nonincreasing order of release time. Let I_j denote the corresponding interference set and $[s_j, t]$ the corresponding interval in Proposition 1.9. As in Proposition 1.11, let w_i denote the elapsed time of a job i under SETF at time t. All processing of the jobs in I_j (by SETF or SRPT) up to time t occurs in this interval, and all processing by SETF in this interval is of jobs in I_j (Proposition 1.10). Thus, the value w_i is precisely the amount of time devoted by SETF to the job i in the interval $[s_j, t]$.

During the interval $[s_j, t]$, the SRPT algorithm (with a unit-speed machine) spends at most $t - s_j$ time processing jobs, and in particular at most $t - s_j$ time processing jobs of I_j. Meanwhile, the SETF algorithm works continually over the interval $[s_j, t]$; at all times $s \in [s_j, t]$ there is at least one active job (Proposition 1.9), and the SETF algorithm never idles with an active job. Thus SETF (with a $(1+\epsilon)$-speed machine) processes $(1+\epsilon)(t - s_j)$ units worth of jobs in this interval, and all of this work is devoted to jobs of I_j (Proposition 1.10).

Now group the jobs of I_j into three categories:

1. Jobs $i \in I_j$ that belong to X_t^* (i.e., SRPT has not completed i by time t).
2. Jobs $i \in I_j$ that belong to X_t but not X_t^* (i.e., SETF has not completed i by time t, but SRPT has).
3. Jobs $i \in I_j$ that belong to neither X_t nor X_t^* (i.e., both SETF and SRPT have completed i by time t).

The SRPT algorithm spends at least as much time as SETF in the interval $[s_j, t]$ processing category-2 jobs (as the former completes them and the latter does not), as per the first step of the intuition in Section 1.4.2. Both algorithms spend exactly the same amount of time on category-3 jobs in this interval (namely, the sum of the processing times of these jobs). We can therefore conclude that the excess time $\epsilon(t - s_j)$ spent by the SETF algorithm (beyond that spent by SRPT) is devoted entirely to category-1 jobs—the jobs of X_t^* (cf., the second step of the outline in Section 1.4.2). We summarize our progress so far in a proposition.

Proposition 1.12 *For every $j = 1, 2, \ldots, k$,*

$$\sum_{i \in I_j \cap X_t^*} w_i \geq \epsilon \cdot (t - s_j).$$

The sum in Proposition 1.12 is, at least, over the jobs $\{1, 2, \ldots, j\}$.

Proposition 1.13 *For every $j = 1, 2, \ldots, k$, the interference set I_j includes the jobs $\{1, 2, \ldots, j\}$.*

Proof Recall that the jobs $\{1, 2, \ldots, k\}$ of $X_t \setminus X_t^*$ are sorted in nonincreasing order of release time. Each job $i = 1, 2, \ldots, j-1$ is released after job j and before job j completes (which is at time t or later), and interferes with j at the time of its release (as SETF begins processing it immediately). □

Combining Propositions 1.12 and 1.13, we can associate unfinished work at time t for SETF with that of SRPT:

Corollary 1.14 *For every $j = 1, 2, \dots, k$,*

$$\sum_{i \in I_j \cap X_t^*} w_i \geq \epsilon \cdot \sum_{\ell=1}^{j} w_\ell.$$

For example, taking $j = 1$, we can identify ϵw_1 units of time that SETF spends processing the jobs of $I_1 \cap X_t^*$ before time t. Similarly, taking $j = 2$, we can identify ϵw_2 different units of time that SETF spends processing the jobs of $I_2 \cap X_t^*$: Corollary 1.14 ensures that the total amount of time so spent is at least $\epsilon w_1 + \epsilon w_2$, with at most ϵw_1 of it already accounted for in the first step. Continuing with $j = 3, 4, \dots, k$, the end result of this process is a collection $\{\alpha(j,i)\}$ of nonnegative "charges" from jobs j of $X_t \setminus X_t^*$ to jobs i of X_t^* that satisfies the following properties:

1. For every $j = 1, 2, \dots, k$, $\sum_{i \in X_t^*} \alpha(j,i) = \epsilon w_j$.
2. For every $i \in X_t^*$, $\sum_{j=1}^{k} \alpha(j,i) \leq w_i$.
3. $\alpha(j,i) > 0$ only if $i \in I_j \cap X_t^*$.

Combining the third property with Proposition 1.11:

$$w_i \leq w_j \text{ whenever } \alpha(j,i) > 0. \tag{1.4}$$

We can extract from the $\alpha(j,i)$'s a type of network flow in a bipartite graph with vertex sets $X_t \setminus X_t^*$ and X_t^*. Precisely, define the flow f_{ji}^+ outgoing from $j \in X_t \setminus X_t^*$ to $i \in X_t^*$ by

$$f_{ji}^+ = \frac{\alpha(j,i)}{w_j}$$

and the flow f_{ji}^- incoming to i from j by

$$f_{ji}^- = \frac{\alpha(j,i)}{w_i}.$$

If we think of each vertex h as having a capacity of w_h, then f_{ji}^+ (respectively, f_{ji}^-) represents the fraction of j's capacity (respectively, i's capacity) consumed by the charge $\alpha(j,i)$. Property (1.4) implies that the flow is expansive, meaning that

$$f_{ji}^+ \leq f_{ji}^-$$

for every j and i.

The first property of the $\alpha(j,i)$'s implies that there are ϵ units of flow outgoing from each $j \in X_t \setminus X_t^*$, for a total of $\epsilon \cdot |X_t \setminus X_t^*|$. The second property implies that there is at most one unit of flow incoming to each $i \in X_t^*$, for a total of at most $|X_t^*|$. Because the flow is expansive, the total amount of flow incoming to X_t^* is at least that outgoing from $X_t \setminus X_t^*$, and so

$$|X_t^*| \geq \epsilon \cdot |X_t \setminus X_t^*|.$$

This completes the proof of Lemma 1.8:

$$|X_t| \leq |X_t^*| + |X_t \setminus X_t^*| \leq |X_t^*| \cdot \left(1 + \frac{1}{\epsilon}\right).$$

1.5 Loosely Competitive Algorithms

An online algorithm with a good resource augmentation guarantee is usually "loosely competitive" with the offline optimal algorithm, which roughly means that, for every input, its performance is near-optimal for most resource levels (cf., Figure 1.2(b)). We illustrate the idea using the online paging problem from Section 1.1; Exercise 1.6 outlines an analogous result in the selfish routing model of Section 1.3.

There is simple and accurate intuition behind the main result of this section. Consider a page request sequence z and a cache size k. Suppose the number of page faults incurred by the LRU algorithm is roughly the same—within a factor of 2, say—with the cache sizes k and $2k$. Theorem 1.3, with $2k$ and k playing the roles of k and h, respectively, then immediately implies that the number of page faults incurred by the LRU algorithm with cache size k is at most a constant (roughly 4) times that incurred by the offline optimal algorithm with the same cache size. In other words, in this case the LRU algorithm is competitive in the traditional sense (Definition 1.2). Otherwise, the performance of the LRU algorithm improves rapidly as the cache size is expanded from k to $2k$. But because 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 a bounded number of different cache sizes.

Here is the precise statement, followed by discussion and a proof.

Theorem 1.15 (Young (2002)) *For every $\epsilon, \delta > 0$ and positive integer n, for every page request sequence z, for all but a δ fraction of the cache sizes k in $\{1, 2, \ldots, n\}$, the LRU algorithm satisfies either:*

1. $\textsc{Perf}(LRU, k, z) = O(\frac{1}{\delta} \log \frac{1}{\epsilon}) \cdot \textsc{Perf}(FIF, k, z)$*; or*
2. $\textsc{Perf}(LRU, k, z) \leq \epsilon \cdot |z|$.

Thus, for every page request sequence z, each cache size k falls into one of three cases. In the first case, the LRU algorithm with cache size k is competitive in the sense of Definition 1.2, with the number of page faults incurred at most a constant (i.e., $O(\frac{1}{\delta} \log \frac{1}{\epsilon})$) times the minimum possible. In the second case, the LRU algorithm has a page fault rate of at most ϵ, and thus has laudable performance in an absolute sense. In the third case neither good event occurs, but fortunately this happens for only a δ fraction of the possible cache sizes.

The parameters δ, ϵ, and n in Theorem 1.15 are used in the analysis only—no "tuning" of the LRU algorithm is needed—and Theorem 1.15 holds simultaneously for all choices of these parameters. The larger the fraction δ of bad cache sizes or the absolute performance bound ϵ that can be tolerated, the better the relative performance guarantee in the first case.

In effect, Theorem 1.15 shows that a resource augmentation guarantee like Theorem 1.3—an apples vs. oranges comparison between an online algorithm with a big cache and an offline algorithm with a small cache—has interesting implications for online algorithms even compared with offline algorithms with the same cache size. This result dodges the lower bound on the competitive ratio of the LRU algorithm (Section 1.1.3) in two ways. First, Theorem 1.15 offers guarantees only for most choices of the cache size k; LRU might perform poorly for a few unlucky cache sizes. This is a reasonable relaxation, given that we don't expect actual page request sequences to be adversarially tailored to the choice of cache size. Second, Theorem 1.15 does not insist on good performance relative to the offline optimal algorithm—good absolute performance (i.e., a very small page fault rate) is also acceptable, as one would expect in a typical application.[9]

We proceed to the proof of Theorem 1.15, which follows closely the intuition laid out at the beginning of the section.

Proof Fix a request sequence z and values for the parameters δ, ϵ, and n. Let b be a positive integer, to be chosen in due time. The resource augmentation guarantee in Theorem 1.3 states that, ignoring additive terms,

$$\textsc{Perf}(LRU, k+b, z) \leq \frac{k+b}{b+1} \cdot \textsc{Perf}(FIF, k, z), \tag{1.5}$$

where $k+b$ and k are playing the roles of k and h in Theorem 1.3, respectively.

There are two cases, depending on whether

$$\textsc{Perf}(LRU, k+b, z) \geq \frac{1}{2} \cdot \textsc{Perf}(LRU, k, z) \tag{1.6}$$

or

$$\textsc{Perf}(LRU, k+b, z) < \frac{1}{2} \cdot \textsc{Perf}(LRU, k, z).$$

Call a cache size k *good* or *bad* according to whether it belongs to the first or second case, respectively. For good cache sizes k, chaining together the inequalities (1.5) and (1.6) shows that

$$\textsc{Perf}(LRU, k, z) \leq 2 \cdot \frac{k+b}{b+1} \cdot \textsc{Perf}(FIF, k, z), \tag{1.7}$$

and hence LRU is competitive (with ratio $\frac{2(k+b)}{b+1}$) in the sense of Definition 1.2.

Consider the set of bad cache sizes; for every such size, adding b extra pages to the cache decreases the number of page faults incurred by the LRU algorithm on z by at least a factor of 2. If there are at least ℓ bad cache sizes between 1 and $t-b$

[9] This may seem like an obvious point, but such appeals to good absolute performance are uncommon in the analysis of online algorithms.

for some t, then we can find ℓ/b bad cache sizes $k_1 < k_2 < \cdots < k_{\ell/b}$ in this interval that are each at least b apart (by taking every bth bad cache size).[10] In this case, using that $\textsc{Perf}(LRU, k, z)$ is nonincreasing in k (Exercise 1.1), we have

$$\textsc{Perf}(LRU, k_{i+1}, z) < \frac{1}{2} \cdot \textsc{Perf}(LRU, k_i, z)$$

for each $i = 1, 2, \ldots, \ell/b$, where $k_{(\ell/b)+1}$ should be interpreted as $k_{\ell/b} + b \leq t$. Chaining all of these inequalities together yields

$$\textsc{Perf}(LRU, t, z) < 2^{-\ell/b} \cdot \textsc{Perf}(LRU, 1, z).$$

Thus, once

$$\ell \geq b \cdot \log_2 \tfrac{1}{\epsilon}, \tag{1.8}$$

we have a page fault rate of at most ϵ:

$$\textsc{Perf}(LRU, t, z) \leq \epsilon \cdot |z|, \tag{1.9}$$

where $|z|$ is the length of the request sequence z.

The time has come to instantiate the parameter b. Guided by our desire to have δn bad cache sizes between 1 and some number t force the condition that $\textsc{Perf}(LRU, k, z) \leq \epsilon|z|$ for all cache sizes $k \geq t$, we take $\ell = \delta n$. The inequality (1.8) then suggests taking $b = \delta n / \log_2 \frac{1}{\epsilon}$.

Cache sizes now fall into three categories:

1. Good cache sizes. By the inequality (1.7) and our choice of b,

   $$\textsc{Perf}(LRU, k, z) = O(\tfrac{1}{\delta} \log \tfrac{1}{\epsilon}) \cdot \textsc{Perf}(FIF, k, z)$$

   for every such cache size k.
2. The smallest δn bad cache sizes in $\{1, 2, \ldots, n\}$. There is no performance guarantee for these cache sizes.
3. Bad cache sizes that are bigger than at least δn other bad cache sizes. Our choices of ℓ and b ensure that the inequality (1.9) holds for such a cache size k, with

   $$\textsc{Perf}(LRU, k, z) \leq \epsilon|z|.$$

Cache sizes in the first and third categories meet the first and second guarantees, respectively, of Theorem 1.15. Cache sizes in the second category constitute at most a δ fraction of the possible cache sizes, so the proof is complete. $\square$

[10] For clarity, we omit the appropriate ceilings and floors from fractions such as ℓ/b.

1.6 Notes

Resource augmentation was first stressed as a first-order analysis framework by Kalyanasundaram and Pruhs (2000), although there were compelling examples much earlier (such as Theorem 1.3, which was proved by Sleator and Tarjan (1985)). The phrase "resource augmentation" was proposed shortly thereafter, by Phillips et al. (2002).

The competitive analysis of online algorithms, including the model and results in Section 1.1, was developed by Sleator and Tarjan (1985). A good general reference for the topic is the book by Borodin and El-Yaniv (1998). Theorem 1.1 is due to Bélády (1967). See Young (1991, §2.4) for empirical comparisons of the FIF, LRU, and FIFO cache replacement policies on benchmark page request sequences.

The selfish routing model described in Section 1.3 was defined by Wardrop (1952). Existence and uniqueness of equilibrium flows (see footnote 6) was proved by Beckmann et al. (1956); see also Roughgarden (2007). The price of anarchy was defined, in a different context, by Koutsoupias and Papadimitriou (1999). Theorem 1.4 and the extension in Exercise 1.4 were proved by Roughgarden and Tardos (2002). The consequent loosely competitive bound (Exercise 1.6) was proved by Friedman (2004).

Pruhs et al. (2004) is a good reference on the competitive analysis of online scheduling algorithms; it includes a figure that inspired Figure 1.2. The optimality of SRPT (Exercise 1.7) was first proved by Schrage (1968). Theorem 1.7 is by Kalyanasundaram and Pruhs (2000), as is Exercise 1.9. One solution to Exercise 1.8 appears in Motwani et al. (1994). There are several more recent and sophisticated resource augmentation guarantees for more complex scheduling problems, for example with multiple machines, jobs with different priorities, and preemptions replaced by a small number of rejections. Good entry points to this literature include Im et al. (2011), Anand et al. (2012), and Thang (2013).

The concept of a loosely competitive online algorithm is due to Young (1994) and Theorem 1.15 is from Young (2002).

Acknowledgments

I thank Jérémy Barbay, Feder Fomin, Kirk Pruhs, Nguyen Kim Thang, and Neal Young for helpful comments on a preliminary draft of this chapter.

References

Anand, S., Garg, N., and Kumar, A. 2012. Resource Augmentation for Weighted Flow-Time Explained by Dual Fitting. Pages 1228–1241 of: *Proceedings of* *the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms (SODA).*

Beckmann, M. J., McGuire, C. B., and Winsten, C. B. 1956. *Studies in the Economics of Transportation.* Yale University Press.

Bélády, L. A. 1967. A Study of Replacement Algorithms for a Virtual Storage Computer. *IBM Systems Journal*, **5**(2), 78–101.

Borodin, A., and El-Yaniv, R. 1998. *Online Computation and Competitive Analysis.* Cambridge University Press.

Friedman, E. J. 2004. Genericity and Congestion Control in Selfish Routing. Pages 4667–4672 of: *Proceedings of the 43rd Annual IEEE Conference on Decision and Control (CDC).*

Im, S., Moseley, B., and Pruhs, K. 2011. A Tutorial on Amortized Local Competitiveness in Online Scheduling. *SIGACT News*, **42**(2), 83–97.

Kalyanasundaram, B., and Pruhs, K. 2000. Speed Is as Powerful as Clairvoyance. *Journal of the ACM*, **47**(4), 617–643.

Koutsoupias, E., and Papadimitriou, C. H. 1999. Worst-case Equilibria. Pages 404–413 of: *Proceedings of the 16th Annual Symposium on Theoretical Aspects of Computer Science (STACS).*

Motwani, R., Phillips, S., and Torng, E. 1994. Nonclairvoyant Scheduling. *Theoretical Computer Science*, **130**(1), 17–47.

Phillips, C. A., Stein, C., Torng, E., and Wein, J. 2002. Optimal Time-Critical Scheduling via Resource Augmentation. *Algorithmica*, **32**(2), 163–200.

Pruhs, K., Sgall, J., and Torng, E. 2004. Online Scheduling. Chap. 15 of: *Handbook of Scheduling: Algorithms, Models, and Performance Analysis.* CRC Press.

Roughgarden, T. 2007. Routing Games. Chap. 18, pages 461–486 of: Nisan, N., Roughgarden, T., Tardos, É., and Vazirani, V. (eds), *Algorithmic Game Theory.* Cambridge University Press.

Roughgarden, T., and Tardos, É. 2002. How Bad Is Selfish Routing? *Journal of the ACM*, **49**(2), 236–259.

Schrage, L. 1968. A Proof of the Optimality of the Shortest Remaining Processing Time Discipline. *Operations Research Letters*, **16**(3), 687–690.

Sleator, D. D., and Tarjan, R. E. 1985. Amortized Efficiency of List Update and Paging Rules. *Communications of the ACM*, **28**(2), 202–208.

Thang, N. K. 2013. Lagrangian Duality in Online Scheduling with Resource Augmentation and Speed Scaling. Pages 755–766 of: *21st Annual European Symposium on Algorithms (ESA).*

Wardrop, J. G. 1952. Some Theoretical Aspects of Road Traffic Research. Pages 325–378 of: *Proceedings of the Institute of Civil Engineers, Pt. II*, vol. 1.

Young, N. 2002. On-Line File Caching. *Algorithmica*, **33**(3), 371–383.

Young, N. E. 1991. *Competitive Paging and Dual-Guided Algorithms for Weighted Caching and Matching.* Ph.D. thesis, Princeton University, Department of Computer Science.

Young, N. E. 1994. The k-Server Dual and Loose Competitiveness for Paging. *Algorithmica*, **11**(6), 525–541.

Exercises

1.1 Prove that for every cache size $k \geq 1$ and every page sequence z,

$$\textsc{Perf}(LRU, k+1, z) \leq \textsc{Perf}(LRU, k, z).$$

1.2 Prove that Theorems 1.3 and 1.15 hold also for the FIFO caching policy.

1.3 Prove a lower bound for all deterministic online algorithms that matches the upper bound for LRU in Theorem 1.3. That is, for every choice of k and $h \leq k$, every constant $\alpha < \frac{k}{k-h+1}$, and every deterministic online paging algorithm A, there exist arbitrarily long sequences z such that $\textsc{Perf}(A, k, z) > \alpha \cdot \textsc{Perf}(FIF, h, z)$.

1.4 Consider a *multicommodity* selfish routing network $G = (V, E)$, with source vertices $s_1, s_2, \ldots, s_k$, sink vertices $t_1, t_2, \ldots, t_k$, and traffic rates $r_1, r_2, \ldots, r_k$. A flow now routes, for each $i = 1, 2, \ldots, k$, r_i units of traffic from s_i to t_i. In an equilibrium flow f, all traffic from s_i to t_i travels on s_i-t_i paths P with the minimum-possible length $\sum_{e \in P} c_e(f_e)$, where f_e denotes the total amount of traffic (across all source-sink pairs) using edge e.

State and prove a generalization of Theorem 1.4 to multicommodity selfish routing networks.

1.5 Deduce Corollary 1.5 from Theorem 1.4.

1.6 This problem derives a loosely competitive-type bound from a resource augmentation bound in the context of selfish routing (Section 1.3). Let $\pi(G, r)$ denote the ratio of the costs of equilibrium flows in G at the traffic rates r and $r/2$. By Theorem 1.4, the price of anarchy in the network G at rate r is at most $\pi(G, r)$.

(a) Use Theorem 1.4 to prove that, for every selfish routing network G and traffic rate $r > 0$, and for at least an α fraction of the traffic rates $\hat{r}$ in $[r/2, r]$, the price of anarchy in G at traffic rate $\hat{r}$ is at most $\beta \log \pi(G, r)$ (where $\alpha, \beta > 0$ are constants, independent of G and r).

(b) Prove that for every constant $K > 0$, there exists a network G with nonnegative, continuous, and nondecreasing edge cost functions and a traffic rate r such that the price of anarchy in G is at least K for every traffic rate $\hat{r} \in [r/2, r]$.

[Hint: use a network with many parallel links.]

1.7 Prove that the shortest remaining processing time (SRPT) algorithm is an optimal algorithm for the problem of scheduling jobs on a single machine (with preemption allowed) to minimize the total flow time.

1.8 Prove that for every constant $c > 0$, there is no non-clairvoyant deterministic online algorithm that always produces a schedule with total flow time at most c times that of the optimal (i.e., SRPT) schedule.

1.9 Consider the objective of minimizing the maximum idle time of a job, where the *idle time* of job j in a schedule is $C_j - r_j - \frac{p_j}{s}$, where C_j is the job's completion time, r_j is its release time, p_j is its processing time, and s is the machine speed. Show that the maximum idle time of a job under the SETF algorithm with a $(1+\epsilon)$-speed machine is at most $\frac{1}{\epsilon}$ times that in an optimal offline solution to the problem with a unit-speed machine.

[Hint: Start from Proposition 1.11.]