Chapter 7

Online

In many application domains events are *not* Poisson distributed. For some applications it even makes sense to (more or less) assume that events are distributed in the worst possible way, e.g. in networks, packets often arrive in bursts. In this section we study discrete event systems from a *worst-case* perspective. In particular, we analyze the price of not being able to foresee the future. This is a phenomenon that often occurs in discrete event systems, but also in our daily life. The analysis tool is often referred to as *Online Theory* or *Online Algorithms*.

7.1 Ski Rental

We start out with a seasonal "toy example," ski rental. Imagine that you want to start a new hobby (e.g. skiing, skateboarding, having a boy- or girlfriend), but you don't yet know whether you will like it. The equipment is expensive, therefore you decide to first rent it a few times, before you buy (or get married!). When dealing with this problem, we (informally speaking) assume that Murphy's law will strike: as soon as you buy, you will lose interest in the subject. Arguments like "I rented skis 17 times, and like it so much that I will go skiing for at least 1717 more times" do not count in Murphy's world. Instead, once you buy skis you can be sure to meet new friends, and they think that skiing is for losers, and snowboarding or whatever is the new hip thing.

We first radically simplify the problem (to make it mathematically more elegant and tractable):

Definition 7.1 (Ski Rental)**.** *The ski rental problem consists of two values:*

- *Input: a real number u, representing the time a skier will end up skiing ($u \geq 0$), chosen by an adversary.*
- *Algorithm: a real number z, at which the algorithm will stop renting skis, and instead buys skis for price 1.*

Remarks:

- The algorithm does not know the discrete event, i.e., the input u.
- The algorithm is represented by a single value. This is rather unusual.
- The cost of the algorithm with value z on input u is $cost_z(u)$:

$$cost_z(u) = \begin{cases} u & \text{if } u \leq z \\ z+1 & \text{if } u > z \end{cases}$$

- The goal is to develop an algorithm z that is good for *any* input u. For determining how "good" an algorithm is, we compare the cost of the algorithm with the cost of an optimal clairvoyant ("offline") algorithm, i.e., an algorithm that knows the future (which in our case corresponds to knowing input u). The cost of the optimal offline algorithm is given by

$$cost_{opt}(u) = \left\{ \begin{array}{ll} u & \text{if } u \leq 1 \\ 1 & \text{if } u > 1 \end{array} \right\} = \min(u, 1).$$

Definition 7.2 (Competitive Analysis)**.** *An online algorithm A is r-competitive if for all finite input sequences I*

$$cost_A(I) \leq r \cdot cost_{opt}(I) + k,$$

where $cost_A$ and $cost_{opt}$ are the cost functions of the algorithm A and the optimal offline algorithm, respectively, and k is a constant independent of the input.

Remarks:

- The smaller we can make r, the better.
- Our parameter r may be a constant, but it can also depend on the input, e.g., $r \in O(\log n)$ or $r \in O(n)$ where n is the size of the input.

Definition 7.3 (Competitive Ratio)**.** *If $k = 0$ in Definition 7.2, then the online algorithm is called **strictly** r-competitive. In this case, the worst-case ratio between the cost of the online and the cost of the optimal offline algorithm, called **competitive ratio**, is often considered directly. Formally, the competitive ratio is defined as*

$$r = \sup_{I \in \mathcal{I}} \frac{cost_A(I)}{cost_{opt}(I)}$$

where $\mathcal{I}$ is the set of all finite input sequences I.

Remarks:

- We take the supremum and not simply the maximum since there are cases where the ratio between the cost of the online and the cost of the optimal offline algorithm is smaller than some specific value for any finite input sequence, but gets arbitrarily close to this value (e.g., you may get closer and closer to this value by making the input sequence longer and longer).

- If an online algorithm has a competitive ratio of r, then it is strictly r-competitive (but not necessarily the other way around!).

Theorem 7.4. *Ski rental is strictly 2-competitive. The best algorithm is $z = 1$.*

Proof. Let us investigate $z = 1$ in the ski rental algorithm. Then,

$$\frac{cost_z(u)}{cost_{opt}(u)} =$$

Cases	$u \leq z = 1$	$u > z = 1$
$u \leq 1$	$\frac{u}{u}$	impossible
$u > 1$	impossible	$\frac{1+1}{1}$

Thus, the worst case is $u > z = 1$, and the competitive ratio is 2. Is this optimal?

- Let's try $z > 1$: In this case the adversary might/will choose $u = z + \epsilon$. Then, the cost ratio is
$$\frac{cost_z(u)}{cost_{opt}(u)} = \frac{z+1}{1} > 2.$$
- If $z < 1$ then the adversary will also choose $u = z + \epsilon$, for some small enough ϵ. Then
$$\frac{cost_z(u)}{cost_{opt}(u)} = \frac{z+1}{u} = \frac{z}{u} + \frac{1}{u} > 2.$$

□

Remarks:

- Everything solved?!? It seems that the algorithm has a big handicap. We assume that the adversary knows every bit about the algorithm (similar to models used in cryptography). The adversary can always present an input which is (arbitrarily close to) worst-case for the algorithm. Maybe the algorithm may decide randomly, rendering the adversary's life more difficult.

7.2 Randomized Ski Rental

Let's look at an algorithm A that chooses randomly between two values, z_1 and z_2 (with $z_1 < z_2$), with probabilities p_1 and $p_2 = 1 - p_1$, respectively.

Algorithm 7.5 Two Value Ski Rental

```
1: throw coin that shows heads with probability p_1
2: if coin shows heads then
3:     buy skis at time z_1
4: else
5:     buy skis at time z_2
6: end if
```

Remarks:

- The cost of Algorithm 7.5 is given by

$$cost_A(u) = \begin{cases} u & \text{if } u \leq z_1 \\ p_1 \cdot (z_1 + 1) + p_2 \cdot u & \text{if } z_1 < u \leq z_2 \\ p_1 \cdot (z_1 + 1) + p_2 \cdot (z_2 + 1) & \text{if } z_2 < u \end{cases}$$

- The adversary, being very evil, will still choose the worst possible input. Note that only $u_1 = z_1 + \epsilon$ and $u_2 = z_2 + \epsilon$ are sensible, as any other input value is inferior to either u_1 or u_2.

Example 7.6. *Consider the algorithm given by the values*

$$z_1 = 1/2, z_2 = 1.$$

What is the best choice for p_1, i.e., the choice that minimizes the competitive ratio?

For $\epsilon \to 0$, we have $cost_A = p_1(z_1 + 1) + p_2 z_1$ if the adversary chooses the input $u_1 = z_1 + \epsilon$, and $cost_A = p_1(z_1 + 1) + p_2(z_2 + 1)$ if the adversary chooses the input $u_2 = z_2 + \epsilon$.

Using the values from above and the fact that $p_2 = 1 - p_1$, we obtain

$$cost_A = p_1 + \frac{1}{2}$$

and

$$cost_A = 2 - \frac{1}{2} \cdot p_1,$$

respectively.

Hence, for the input $u_1 = z_1 + \epsilon$, we have (for $\epsilon \to 0$)

$$\frac{cost_A}{cost_{opt}} = 2p_1 + 1,$$

and for the input $u_2 = z_2 + \epsilon$, we have

$$\frac{cost_A}{cost_{opt}} = 2 - \frac{1}{2} \cdot p_1.$$

As the adversary will pick the larger of the two ratios by choosing the respective input, the optimal choice for p_1 (for the online algorithm) minimizes the maximum of the two ratios. Since one of the two ratios, seen as a function of the parameter p_1, is monotonically increasing while the other one is monotonically decreasing, the minimum is assumed when both ratios have the same value. Setting

$$2p_1 + 1 = 2 - \frac{1}{2} \cdot p_1$$

yields

$$p_1 = \frac{2}{5},$$

which implies a competitive ratio of

$$\frac{cost_A}{cost_{opt}} = \frac{9}{5}.$$

Remarks:

- In other words, for this particular randomized algorithm, the expected competitive ratio is only 1.8, below the best possible deterministic algorithm. Mind, however, that this new bound is in expectation only!
- Maybe one can do *even* better by allowing the algorithm to choose more than two values? Maybe even *infinitely* many values?!? A scenario is shown in Figure 7.7.

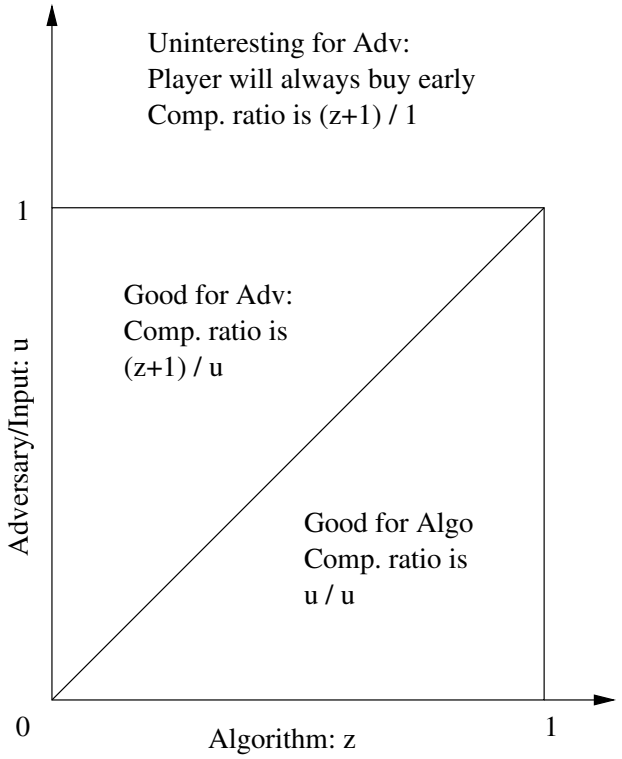

Figure 7.7: Choosing more than two values

- Then, one might think, the expected competitive ratio is

$$E[r] = \frac{1}{2} + \int_0^1 \int_0^u \frac{z+1}{u} dz du = \ldots = 1.75.$$

- Was that a valid argument?
- No. We assumed that the adversary chooses u with uniform distribution. This is not okay. In this specific example, an adversary can cause much more harm by choosing values close to 1. In addition, it was not correct to sum up the ratios of the costs, instead we should compute the ratio of the expected costs.

- Instead, we should rather solve

$$E[r] = \frac{\int_0^1 \int_0^u (z+1)p(z)d(u)dzdu + \int_0^1 \int_u^1 up(z)d(u)dzdu}{\int_0^1 ud(u)du},$$

  where $p(z)$ is the probability distribution of the algorithm, and $d(u)$ is the probability distribution of the adversary, with $\int p(z) = \int d(u) = 1$. The adversary chooses its distribution $d(u)$ such that it maximizes the expected competitive ratio $E[r]$, and the algorithm chooses its distribution $p(z)$ such that it minimizes $E[r]$. Note that the adversary chooses its distribution *after* the algorithm chooses its distribution.

- This is a very hard task. However, does it really make sense for the adversary to choose the input according to a probability distribution, as it did in the previous example? As it turns out, the adversary cannot do better than choosing one (worst) input value deterministically, i.e., it does not gain anything from using randomization.

- Recall that the adversary chooses the input *after* the online algorithm chooses its probability distribution $p(z)$. Hence, in order for our algorithm to be strictly r-competitive, it is necessary that the cost of the online algorithm is at most r times the cost of the optimal offline algorithm, for any input u. However, achieving the latter is also sufficient in order to be r-competitive! In other words, all we have to do for a competitive ratio of r is to choose the algorithm's probability function $p(z)$ such that $cost_A(u) \leq r \cdot cost_{opt}(u)$ for all u. With this approach, we can actually show the following theorem.

Theorem 7.8. *The online algorithm given by the probability distribution* $p(z) = \frac{e^z}{e-1}$ *is strictly* $\frac{e}{e-1}$*-competitive in expectation.*

Proof. For a better understanding of the particular choice of our algorithm, let's have a look at the steps leading to this choice. Recall that the algorithm's cost is

$$cost_z(u) = \begin{cases} u & \text{if } u \leq z \\ z+1 & \text{if } u > z \end{cases}$$

Again, it seems natural to restrict the algorithm to values between 0 and 1. Also the adversary can restrict itself to values between 0 and 1, because, if a value higher than 1 is presented, the adversary and the algorithm infer exactly the same cost as if the value 1 was presented. Therefore,

$$\int_0^u (z+1)p(z)dz + \int_u^1 u \cdot p(z)dz \leq r \cdot u, \text{ with } \int_0^1 p(z)dz = 1.$$

Having a hunch that the best probability function will probably be an equality, we immediately try

$$\int_0^u (z+1)p(z)dz + u\int_u^1 p(z)dz = r \cdot u, \text{ with } \int_0^1 p(z)dz = 1.$$

We first differentiate with respect to u, getting

$$(u+1)p(u) + \int_u^1 p(z)dz + u \cdot (-p(u)) = p(u) + \int_u^1 p(z)dz = r.$$

We again differentiate with respect to u, and get

$$\frac{\delta p(u)}{\delta u} - p(u) = 0 \Leftrightarrow \frac{\delta p(u)}{\delta u} = p(u).$$

That's one of the few differential equations everybody knows:

$$p(u) = \alpha \cdot e^u.$$

In order to reveal α we use $\int_0^1 p(z)dz = 1$:

$$1 = \int_0^1 \alpha e^z dz = \alpha(e^1 - e^0) \Rightarrow \alpha = \frac{1}{e-1}.$$

In other words, $p(u) = \frac{e^u}{e-1}$. We insert $p(u)$ into the first differentiation:

$$r = p(u) + \int_u^1 p(z)dz = \frac{e^u}{e-1} + \frac{e^1 - e^u}{e-1} = \frac{e}{e-1}.$$

Note that also for inputs $u > 1$ the inequality $cost_A(u) \leq r \cdot cost_{opt}(u) = r \cdot 1$ holds. □

Remarks:

- The big question remains: Can we get any better?!?

7.3 Lower Bounds

Time to think about lower bounds. Lower bounds for randomized algorithms often use the Von Neumann/Yao Principle, which we state and use without proof:

Theorem 7.9 (Von Neumann/Yao Principle)**.** *Choose a distribution over problem instances, e.g. $d(u)$ for ski rental. If for this distribution all deterministic algorithms cost at least r, then r is a lower bound for the best possible randomized algorithm.*

Remarks:

- For ski rental we are in the lucky situation that we can easily parameterize all possible deterministic algorithms, simply by $z \geq 0$. Now we have to choose a distribution of inputs, with $d(u) \geq 0$ and $\int d(u) = 1$.

Example 7.10. *Consider the distribution $d(u) = 1/2$ for $0 \leq u \leq 1$ and $d(\text{“}\infty\text{”}) = 1/2$. Now, let's have a look at two simple online algorithms:*

- *$z = 0$ (immediate buy): incurs a constant cost 1 for all possible input distributions: Therefore $cost_{z=0}(d(u)) = 1$.*
- *$z = 1$ (worst-case deterministic algorithm): incurs the same cost as the optimal offline algorithm for small u but cost 2 for $u = \infty$ which happens with probability $1/2$; when summing up we see that $cost_{z=1}(d(u)) = 5/4$.*

In general, the cost of the optimal offline algorithm is

$$cost_{opt}(d(u)) = \frac{1}{2}\int_0^1 u du + \frac{1}{2}\cdot 1 = \frac{3}{4}.$$

For general $z \leq 1$ the cost of the online algorithm is

$$\begin{aligned} cost_z(d(u)) &= \frac{1}{2}\left(\int_0^z u du + \int_z^1 (z+1) du\right) + \frac{1}{2}(z+1) \\ &= \frac{1}{2}\left(\frac{z^2}{2} + (z+1)(1-z) + (z+1)\right) = 1 + \frac{z}{2} - \frac{z^2}{4} \geq 1. \end{aligned}$$

For general $z > 1$ the cost of the online algorithm is

$$cost_z(d(u)) = \frac{1}{2}\int_0^1 u du + \frac{1}{2}(z+1) = \frac{1}{4} + \frac{z+1}{2} > 5/4.$$

Using $cost_{opt}(d(u)) = 3/4$, we conclude that the competitive ratio r is at least $4/3 \approx 1.33$.

Remarks:

- Note that for distribution $d(u)$ indeed $z = 0$ is the best algorithm.
- The lower bound of 1.33 and the upper bound of 1.58 do not match.
- As argued above, the immediate buy algorithm is worst with very small u. In order to make our lower bound stronger it could therefore be beneficial to tune the input distribution such that it contains more small u values.
- Guessing the right input distribution is not trivial. However, similarly to the upper bound, it can be derived using differential equations. The worst input distribution is $d(u) = 1/e^u$, for $0 < u < 1$, and $d(\text{“}\infty\text{”}) = 1/e$.
- Next, let us study some online problems in the Internet. We will discover surprising connections to ski rental.

7.4 The TCP Acknowledgement Problem

TPC is a layer 4 networking protocol of the Internet. It features, among other things:

- An error handling mechanism which tackles transmission errors and disordering of packets, using sequence numbers and acknowlegdements.
- A “friendly” exponential slow start mechanism such that new connections do not overload the network.
- Flow Control: A sliding window sender/receiver buffer that simplifies handling and prevents the receiver buffer from overload.

- Congestion Control: A backoff mechanism that should prevent network overloading.

We study a single sender/receiver pair, where the sender sends packets and the receiver acknowledges them (without sending packets itself). There are several TCP implementations available, with various acknowledgement-procedures. In order to save resources, no implementation sends acknowledgements right away. One version of Solaris, for example, always waits 50ms before acknowledging in order to support multiple acknowledgements in a single message. In one version of BSD, TCP-Ack has a 200ms heartbeat, and acknowledges all packets received so far. All implementations send cumulative acknowledgements ("I received all packets up to packet x"). This mechanism is the subject of this section.

At the receiver side, the situation looks like in Figure 7.11.

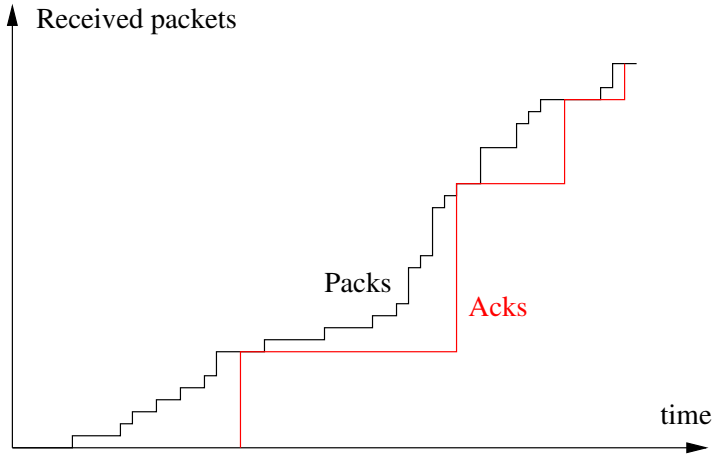

Figure 7.11: TCP ACK problem

Definition 7.12 (TCP Acknowledgement Problem)**.** *The receiver's goal is a scheme which minimizes the number of acknowledgements* ***plus*** *the sum of the latencies for each packet, where the latency of a packet is the time difference from arrival to acknowledgement. More formally, we have*

- *n packet arrivals, at times: $a_1, a_2, \ldots, a_n$*
- *k acknowledgements, at times: $t_1, t_2, \ldots, t_k$*
- *And we want to minimize:*

$$\min k + \sum_{i=1}^{n} latency(i), \text{ with } latency(i) = t_j - a_i, \text{ where } j \text{ such that } t_{j-1} < a_i \leq t_j.$$

Remarks:

- Note that in Figure 7.11 the total latency is exactly the area between the two curves.
- Clearly, we are comparing apples with oranges when comparing the number of acknowledgements with the sum of latencies. However, when scaling time accordingly, this should not be a big problem.
- There are quite a few technical exceptions. In many implementations, signaling packets (e.g. SYN, FIN) are usually acknowledged faster; also the TCP standard wants implementations to acknowledge packets within 500ms. Since the receiver is usually also sender, it might also delay its own sending packets.
- In our studies we do not learn the future from the past. A machine learning approach could give a totally different perspective.
- How might a good algorithm for the TCP Acknowledgement Problem look like?

Algorithm 7.13 $z = 1$ Algorithm

1: Whenever a rectangle with area $z = 1$ does fit between the two curves in Figure 7.14, send an acknowledgement, acknowledging all previous packets.

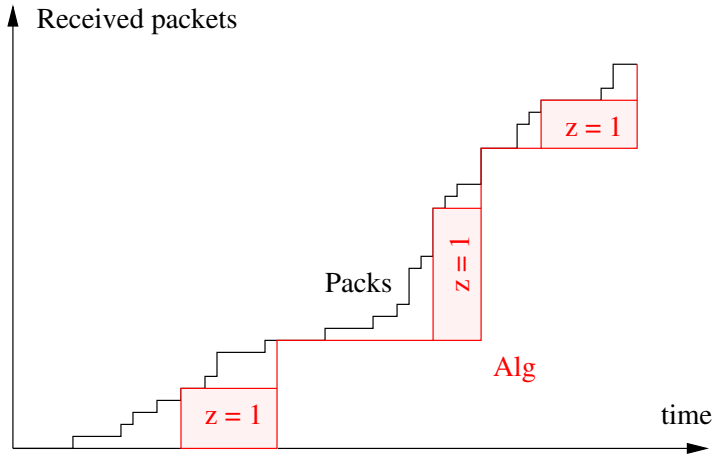

Figure 7.14: The $z = 1$ algorithm

Lemma 7.15. *The optimal algorithm sends an ACK between any pair of consecutive ACKs by Algorithm 7.13.*

Proof. For the sake of contradiction, assume that, among all algorithms who achieve the minimum possible cost, there is no algorithm which sends an ACK between two ACKs of Algorithm 7.13. We propose to send an additional ACK at the beginning (left side) of each $z = 1$ rectangle. Since this ACK saves latency 1,

it compensates the cost of the extra ACK. That is, there is an optimal algorithm who chooses this extra ACK. □

Theorem 7.16. *Algorithm 7.13 is 2-competitive*

Proof. We have $cost_{opt} = k_{opt} + latency_{opt}$ and $cost_{z=1} = k_{z=1} + latency_{z=1}$. Since the optimal algorithm sends at least one ACK between any two consecutive ACKs of Algorithm 7.13, we know $k_{z=1} \leq k_{opt}$.

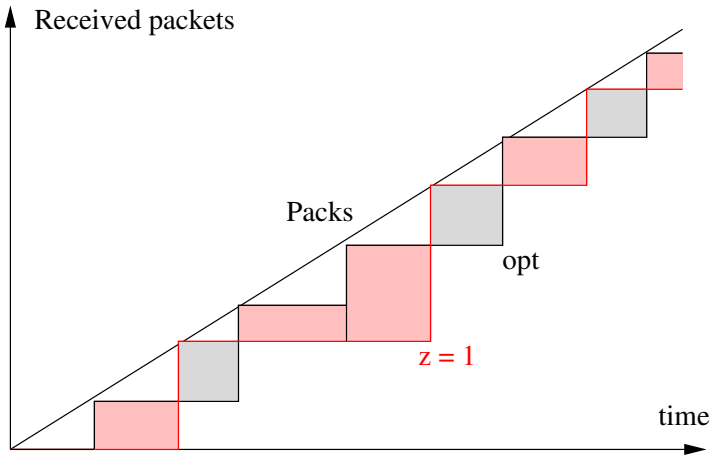

Figure 7.17: Algorithm 7.13 vs. the optimal algorithm

Also, by definition (see Figure 7.17),

$$\begin{aligned} latency_{z=1} &= latency_{opt} + latency(z = 1 \text{ without } opt) - latency(opt \text{ without } z = 1) \\ &\leq latency_{opt} + latency(z = 1 \text{ without } opt). \end{aligned}$$

Using $latency(z = 1 \text{ without } opt) < k_{opt} \cdot 1$ (if any of these rectangles were of size 1 or larger, Algorithm 7.13 would have ACKed earlier) we get:

$$\begin{aligned} cost_{z=1} &= k_{z=1} + latency_{z=1} \\ &\leq k_{opt} + latency_{opt} + latency(z = 1 \text{ without } opt) \\ &< k_{opt} + latency_{opt} + k_{opt} \cdot 1 \\ &= 2 \cdot k_{opt} + latency_{opt} \leq 2 \cdot cost_{opt} \end{aligned}$$

□

Remarks:

- It is no coincidence that we called Algorithm 7.13 the $z = 1$ algorithm. Similarly to ski rental, it is possible to choose any z. In fact, if you *really* think about it, the TCP ACK problem is in fact very much like ski rental! Indeed, if you wait for a rectangle of size z with probability $p(z) = \frac{e^z}{e-1}$, you end up with a randomized TCP ACK solution which is $\frac{e}{e-1}$ competitive in expectation.

- Many other problems are also just like ski rental! That's why we studied it in the first place. E.g. the Halbtax-Problem (originally known as the Bahncard problem). Buying a Halbtax-Card which reduces each trip by β is $\frac{e}{e-1+\beta}$ competitive.

7.5 Competitive Lists with Move-to-Front

Consider a list L containing n items, for example the collection of your favorite records. Whenever an item x in L is requested, the list is scanned from the beginning until x is found. The cost of accessing x is k if x is the k^{th} item in L. In order to better respond to subsequent requests, the position of any two adjacent items in L may be swapped. Such a swap costs 1. Requests to items in L arrive in an online fashion. Here is a simple Move-to-Front algorithm.

Algorithm 7.18 Move-to-Front (M2F)

1: **if** item x is requested **then**
2: Access x and move x to the front.
3: **end if**

Remarks:

- We would like to compare the M2F algorithm with the optimal offline algorithm OPT that knows the entire sequence of requests in advance. Note that the per operation cost of M2F can be arbitrarily worse than OPT. For example, OPT may already move x to the front of L, before requested as OPT knows the request in advance. As such, OPT can serve request x with cost 1, whereas the cost for M2F to serve x may be much higher.

- However, OPT must pay a price for moving x to the front of L at an earlier time. So even though the cost of that one request x may be in favor of OPT, the *total* cost of OPT and M2F may still be comparable.

- *Amortized Analysis* is a method to analyze the total cost of a sequence of operations, e.g., using the *potential method*. A *potential function* Φ is defined to reflect differences in the state of different systems. Instead of just measuring the cost of each operation, we also measure the change in the potential function after each operation. Intuitively, the potential function models the potential cost savings for future operations.

Definition 7.19 (Amortized Cost)**.** *The amortized cost amortized*(op) *of an operation op is defined as*

$$\textit{amortized}(op) := \textit{cost}(op) + \Delta\Phi(op),$$

where cost(op) *is the actual cost of op, and* $\Delta\Phi(op)$ *is the change of potential* Φ *caused by op.*

Remarks:

- Ideally, Φ is defined so that the amortized cost is small, and always the same. Thus, the change in potential should be negative for high cost operations and positive for low cost operations.

Lemma 7.20. *Let $\Phi_{initial}$ and Φ_{final} are the potential values before the first and after the last operation in a sequence S of operations. If $\Phi_{final} \geq 0$, then*

$$\sum_{op \in S} \textit{cost}(op) \leq \sum_{op \in S} \textit{amortized}(op) + \Phi_{initial}.$$

Proof. Using Definition 7.19, the total amortized cost for the operations in S is

$$\begin{aligned} \sum_{op \in S} \text{amortized}(op) &= \sum_{op \in S} \text{cost}(op) + \sum_{op \in S} \Delta\Phi(op) \\ &= \sum_{op \in S} \text{cost}(op) + \Phi_{final} - \Phi_{initial}. \end{aligned}$$

As $\Phi_{final} \geq 0$, the claim follows . □

Theorem 7.21. *M2F algorithm is strictly 4-competitive.*

Proof. Denote by OPT an optimal algorithm. We keep track of two lists L_{M2F} and L_{OPT}, i.e., the list L as it is maintained by M2F and OPT, correspondingly. Initially $L_{M2F} = L_{OPT} = L$. For the two lists L_{M2F} and L_{OPT}, an *inversion* is a pair of items (x, y) which appear in different order in L_{M2F} and L_{OPT}.

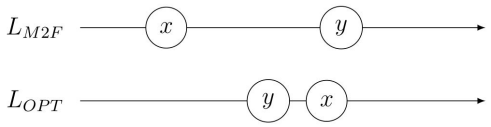

Figure 7.22: The inversion (x, y) between L_{M2F} and L_{OPT}.

We use the following definition of potential function Φ in our analysis.

$$\Phi := 2 \cdot (\text{number of inversions between } L_{M2F} \text{ and } L_{OPT})$$

Initially the potential $\Phi = 0$ since the lists are equal. In every step, Φ is non-negative since the number of inversions is non-negative. Thus the total cost of M2F is upper bounded by the total amortized cost of M2F. It therefore suffices to show that M2F's amortized cost is at most 4 times the cost of OPT.

Fix a sequence of requests and a request r in that sequence, and denote by x the item requested by r. Denote by k and j the position of x in L_{M2F} and L_{OPT} before handling r, respectively.

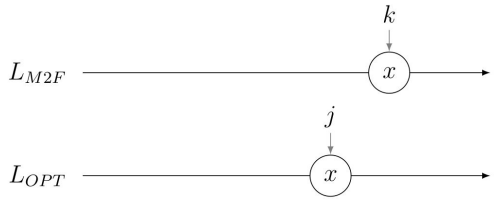

Figure 7.23: Item x in L_{M2F} and L_{OPT} before handling request r.

The cost cost(r) for M2F is k for the initial scan, in addition M2F uses $k-1$ swaps to move x to the front of L, i.e., cost$(r) = 2k-1$. The cost amortized(r) consists of cost(r) and the change in the potential function $\Delta\Phi(r)$. The change of potential after request r has been processed can be determined in two steps: first, the change $\Delta\Phi_{M2F}$ after $M2F$'s list maintenance and second, the change $\Delta\Phi_{OPT}$ after OPT's list maintenance.

Let us first estimate $\Delta\Phi_{M2F}$. Since M2F does not change the relative order of non-requested items, all affected inversions must involve item x. Furthermore x is only swapped with items y that precede x in L_{M2F}. Let y be an item preceding x in L_{M2F} before M2F's list maintenance. We say that item y is *bad* if y precedes x also in L_{OPT}, otherwise y is *good*. If y is bad, then a new inversion is created, otherwise an inversion is destroyed. There are at most $j-1$ bad items, and therefore at least $(k-1)-(j-1)$ good items. Recalling that Φ counts each inversion twice, we conclude that

$$\Delta\Phi_{M2F} \leq 2 \cdot \Big(\big(j-1\big) - \big((k-1)-(j-1)\big) \Big) = 4j - 2k - 2.$$

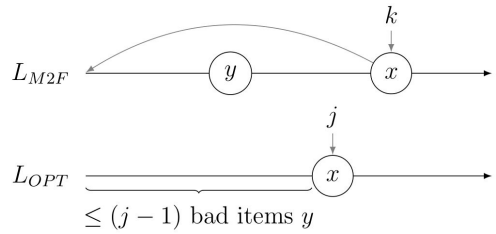

Figure 7.24: Items x, y in L_{M2F} and L_{OPT} before handling request r.

Second, we compute the change $\Delta\Phi_{OPT}$. Denote by s the number of swap-operations performed by OPT while handling request r. Every swap performed by OPT creates at most one new inversion. Thus, the change $\Delta\Phi_{OPT}$ is at most $2s$.

Furthermore, every such swap increases $\text{cost}_{OPT}(r)$ of the optimal algorithm by exactly 1. Recall that the cost for finding item x in L_{OPT} is j, and therefore

$\text{cost}_{OPT}(r) = j + s$. Now, we can bound amortized(r) as

$$\begin{aligned}\text{amortized}(r) &= \text{cost}(r) + \Delta\Phi_{M2F} + \Delta\Phi_{OPT} \\ &\leq 2k - 1 + 4j - 2k - 2 + 2s \\ &= 4j - 3 + 2s \\ &< 4j + 2s \\ &\leq 4 \cdot (j + s) = 4 \cdot \text{cost}_{OPT}(r).\end{aligned}$$

□

7.6 Two Servers on a Line

Lionel and Richie are brothers who work as ice cream vendors at a 1 km long beach. Whenever a tourist wants to buy some ice cream (a "request" event), one of the two brothers must serve the customer. Since the beach is nice and flat, Lionel and Richie always know the exact location of a new request. There are no concurrent requests, i.e., between any two requests there is always enough time to reach the customer and sell the ice cream.

In Figure 7.6, we see an example from the brothers' daily life: The i^{th} request occurs to the right of Richie. In the example, this request is served by Richie, for which he covers a distance of 0.2 kilometers. The $(i+1)^{\text{st}}$ request occurs between Lionel and Richie. Who should serve it?

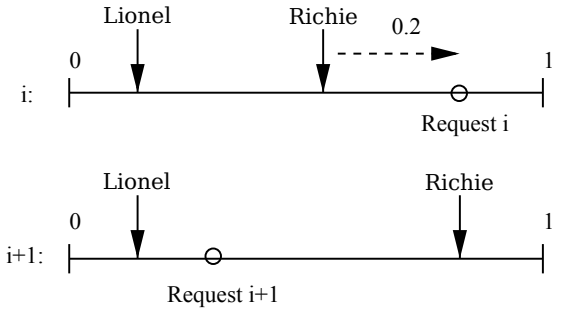

Figure 7.25: "The beach"

Assume that Lionel starts at position 0 and Richie at position 1. A request is defined by its position r_i in the interval $[0, 1]$. Lionel and Richie have to serve a sequence $S = r_1, r_2, \ldots$ of requests.

Let d_L and d_R be the total distances that Lionel resp. Richie cover on a given day. In order to be not too tired in the evening, Lionel and Richie want to keep the total distance they cover together as small as possible, i.e., they want to minimize $d_L + d_R$. What strategy should Lionel and Richie choose?

Algorithm 7.26 Greedy Strategy

1: Each request is served by the brother who is closer to the request

Lemma 7.27. *The cost of the greedy strategy can be arbitrarily close to n times the cost of the optimal solution, where the number of requests is* $2n$.

Proof. Consider the request sequence $S = \frac{1}{2} - \epsilon\ ,\ 0\ ,\ \frac{1}{2} - \epsilon\ ,\ 0\ ,\ \frac{1}{2} - \epsilon\ ,\ \ldots$, for an arbitrarily small constant ϵ. Clearly, all requests of S are handled by Lionel who has to go back and forth between the two points 0 and $\frac{1}{2} - \epsilon$. Hence, the total cost of the simple greedy strategy is $cost_{Greedy} = n \cdot (\frac{1}{2} - \epsilon)$.

In contrast, the optimal solution for sequence S is much better. Richie could move for the first request to position $\frac{1}{2} - \epsilon$, and the two friends could remain at their position forever thereafter. Hence, the optimal cost $cost_{OPT} = \frac{1}{2} + \epsilon$. Thus, the competitive ratio is

$$\frac{cost_{Greedy}}{cost_{OPT}} = \frac{n \cdot (\frac{1}{2} - \epsilon)}{\frac{1}{2} + \epsilon} \approx n\,.$$

□

Remarks:

- Can we do better? Yes, a lot better in fact: There is a deterministic 2-competitive algorithm.

Algorithm 7.28 2-Competive Algorithm

```
1: if request is located to the left of Lionel then
2:     Lionel serves the request
3: else if request is located to the right of Richie then
4:     Richie serves the request
5: else
6:     Both Lionel and Richie move towards the request at equal speed until (at
       least) one of them serves the request
7: end if
```

Theorem 7.29. *Algorithm 7.28 is* 2*-competitive.*

Proof. For a given request sequence, let OPT and ALG denote the optimal solution and the solution computed by our algorithm, respectively. Also, let L^* and R^* be the current positions of Lionel and Richie in the optimal solution OPT. Similarly, we denote by L and R the current positions of Lionel and Richie in ALG.

Note that $L \leq R$. We can also assume that $L^* \leq R^*$ holds, because we can simply "rename" Richie and Lionel when they cross each other in the optimal solution and have the same performance.

We define the following potential function:

$$\Phi \ = \ 2(|L^* - L| + |R^* - R|) + |L - R|.$$

We estimate the change $\Delta\Phi$ after both the OPT and ALG have served the next request in two steps: first, the change $\Delta\Phi_{OPT}$ after OPT has served the request and second, the change $\Delta\Phi_{ALG}$ after ALG has served the request.

Denote by d^* the total distance moved by the brothers in OPT to serve the request. Note that $\Delta\Phi_{OPT} \leq 2d^*$, since the term $|L-R|$ is not changed as only the OPT brothers move, and $|L^* - L| + |R^* - R|$ can increase by at most d^*.

Denote by d the total distance moved by the brothers in ALG to serve the request. We want to show $\Delta\Phi_{ALG} \leq -d$. We distinguish two cases. i) First, assume that the request is either to the left of Lionel or to the right of Richie, i.e., only one of the two moves by a distance of d. In this case, the term $|L-R|$ increases by d. To analyze the change in term $|L^*-L|+|R^*-R|$, assume w.l.o.g. that ALG moves Lionel towards the left to serve the request. As ALG does not move Richie, the term $|R^* - R|$ remains same. In the optimal solution, either Lionel or Richie served the request. If Lionel did, then $|L^* - L|$ decreases by d (from d to 0). If Richie did, then $|L^* - L|$ also decreases by d as $L^* \leq R^*$, and Lionel moves left such that $L = R^*$. Thus, we have $\Delta\Phi_{ALG} \leq -2d + d = -d$.

ii) Second, assume that the request is between Lionel and Richie and each of them move a distance $d/2$ towards the request from opposite sides. Because both brothers move towards each other, the term $|L - R|$ must decrease by d. Now, consider the change in term $|L^* - L| + |R^* - R|$. In the optimal solution, either Lionel or Richie served the request. If Lionel did, then term $|L^* - L|$ decreases by $d/2$ and the term $|R^* - R|$ increases by at most $d/2$ because Richie walked by at most that distance. If Richie did, then similarly the term $|R^* - R|$ decreases by $d/2$ and the term $|L^* - L|$ increases by at most $d/2$. Thus, the term $|L^* - L| + |R^* - R|$ increases by at most 0. In combination with the above observation that $|L - R|$ decreases by d, this proves $\Delta\Phi_{ALG} \leq -d$.

Using the potential function, we can now prove the competitive ratio. We have $\Delta\Phi = \Delta\Phi_{OPT} + \Delta\Phi_{ALG} \leq 2d^* - d$ as shown above, and hence the following for a request r.

$$\text{amortized}(r) = d + \Delta\Phi \;\leq\; 2d^* = 2 \cdot cost_{OPT}(r) \tag{7.1}$$

Denote by $\Phi_{initial}$ and Φ_{final} the initial respectively final potentials. Note that Φ is non-negative and so $\Phi_{final} \geq 0$. Thus,

$$\begin{aligned} cost_{ALG} &= \sum_{r \in S} \text{amortized}(r) + \Phi_{initial} \text{ (with Lemma 7.20)} && (7.2)\\ &\leq \sum_{r \in S} 2 \cdot cost_{OPT}(r) + 1 \text{ (using 7.1)} && (7.3)\\ &\leq 2 \cdot cost_{OPT} + 1\,, \end{aligned}$$

where $cost_{ALG}$ and $cost_{OPT}$ is the total cost of ALG and OPT for serving all the requests. □

Remarks:

- The problem of our two ice cream selling brothers is a special case of a more general problem, the so-called k-server problem, in which there are k agents (or servers) instead of our two brothers, and the requests occur in arbitrary metric spaces, instead of on a line.

7.7 The TCP Congestion Control Problem

As a next example we study the sender side of TCP. We ask: How many segments (or packets, or bytes) per second can a sender inject into the network without overloading it? The problem is that a sender does not know the current bandwidth between itself and the receiver. And, more importantly, this bandwidth might change over time with other connections starting up, or closing down.

Here's our model:

- We divide the time into periods (or slots).
- In each period t there is an unknown threshold u_t, where u_t is the number of packets (or segments, or bytes) that could successfully be transmitted from sender to receiver, without overloading the network.
- In period t, the sender chooses to transmit x_t packets.
- If $x_t \leq u_t$ we are fine. However, sending at too conservative or small rates $x_t \ll u_t$ is a waste of the available bandwidth. One possible way to capture this aspect would be to use an *opportunity cost* function of the form $cost_t = u_t - x_t$.
- If $x_t > u_t$, we are not fine. We are overloading the channel. There are several cost models possible. In a severe cost model, nothing gets transmitted ($cost_t = u_t$), in a less severe cost model, some fraction of the packets might get dropped (e.g. $cost_t = \alpha(x_t - u_t)$).

7.8 The Static Model

We start out with the simplest possible model, where the bandwidth is constant over time, that is, $u_t = u$. The problem is then to find the correct bandwidth u (with something like binary search); once the sender found the correct bandwidth, there will be no more cost. We assume first that u is an integer, and that $1 \leq u \leq n$, that is, there is an upper bound n for the bandwidth.

Possible algorithms:

- Plain old binary search needs $\log n$ search steps. For a worst-case choice of u the algorithm will often inject too many packets, and (in a severe cost model) have cost $u = \Theta(n)$ in most steps, thus the total cost is $\Theta(n \log n)$.
- A standard TCP congestion control mechanism is usually following the AIMD (Additive Increase Multiplicative Decrease) paradigm: Once TCP sends so many packets that the network becomes overloaded, routers will start dropping packets. The sender can witness this (with missing ACKs), and consequently decreases its transmission rate (for example in a multiplicative way, e.g., by a factor 2). Then the sender starts increasing its transmission rate again, but slowly, to approach the "right" bandwidth again (for example by 1, in an additive way). In our model, if the real bandwidth is $u = n - 1$, such an algorithm will clearly be very much off the right bandwidth u most of the time. Since approaching u takes $\Theta(n)$

steps, and in the severe cost model most steps cost $u - x_t = \Theta(n)$, the cost of the AIMD algorithm is $\Theta(n^2)$.

- The obvious question: Can we do better?!?

Algorithm 7.30 Shrink Algorithm: The algorithm operates on a pinning interval $[i, j]$, originally $[i, j] = [1, n]$. It has two phases.

Phase 1: Find the right power-of-two-upper bound, that is, find j such that $2^k < j \leq 2^{k+1}$ by testing $2^k + 1$. If $2^k + 1 \leq u$ goto Phase 2, else set $[i, j] = [1, 2^k]$ and stay in Phase 1.

Phase 2: We are given $[i, j]$ with $2^{t-1} + 1 \leq i < j \leq 2^t$. Now we test

$$i + \max\left(1, \frac{2^t}{2^{2^{m+1}}}\right)$$

with m being the largest integer such that $j - i < \frac{2^t}{2^{2^m}}$. Then adapt $[i, j]$ accordingly.

Remarks:

- It can be shown that the cost of the Shrink algorithm is $O(n \log\log n)$.
- For large n, it is remarkable that the vast majority of increase steps are increments by just 1. And almost all decrease steps are substantial. In other words, the algorithm is an AIMD algorithm.
- If n is not known, we can find an upper bound of u quickly by a repeated squaring technique first, that is, test 2, then $2^2 = 4$, then $4^2 = 16$, then $16^2 = 256, \ldots$. It can be shown that the total cost is $O(u \log\log u)$.
- There is a lower bound of $O(u \log\log u / \log\log\log u)$. Hence the Shrink algorithm is asymptotically almost optimal.
- However, this was only an warm-up example. What we are really interested in are dynamic models.

7.9 The Dynamic Model

In this section, the threshold may vary from step to step, i.e., the adversary chooses a sequence $\{u_t\}$. Thereby, the adversary knows the algorithm's sequence $\{x_t\}$ of probes/tests in advance. Clearly, we are again in the realm of online algorithms and competitive analysis.

We have postulated that $cost_{Alg}(I) \leq r \cdot cost_{opt}(I)$. Observe that an optimal *offline* algorithm knowing the input (as in ski rental or TCP ACK) can always

play $x_t = u_t$, which implies that $cost_{opt} = 0$. No online algorithm can be competitive!

For this reason it seems more fruitful to look at *gain* (or profit) rather than cost. We update our definition from ski rental as follows:

Definition 7.31 (Competitive Analysis)**.** *An online algorithm A is strictly r-competitive if for all finite input sequences I*

$$\begin{aligned} cost_A(I) &\leq r \cdot cost_{opt}(I), \text{ or} \\ r \cdot gain_A(I) &\geq gain_{opt}(I). \end{aligned}$$

Remarks:

- Note that in both cases $r \geq 1$. The closer r is to 1, the better is an algorithm.

For a severe cost model, a natural definition of *gain* could look as follows:

$$gain_{x_t}(u_t) = \begin{cases} x_t & \text{if } x_t \leq u_t \\ 0 & \text{if } x_t > u_t \end{cases}$$

However, note that our adversary is too strong because (knowing the algorithm) it can always present an $u_t < x_t$ (or, if $x_t = 0$, any u_t). The total gain of the algorithm (given as $\sum_t gain_{Alg}(t)$) is 0. We therefore need to further restrict the power of the adversary. Several restrictions seem to be reasonable and interesting:

- Bandwidth in a fixed interval: $u_t \in [a, b]$
- Multiplicatively (or additively) changing bandwidth: $u_t/\mu \leq u_{t+1} \leq \mu \cdot u_t$ (or $u_t - \alpha \leq u_{t+1} \leq u_t + \alpha$)
- Changes with bursts

In the following, the three restrictions will be studied in turn.

7.10 Bandwidth in a Fixed Interval

We start out by letting the adversary choose $u_t \in [a, b]$. The algorithm is aware of the upper bound b and the lower bound a. We first restrict ourselves to deterministic algorithms. In this case, note the following:

- If the deterministic algorithm plays $x_t > a$ in round t, then the adversary plays $u_t = a$.
- Therefore the algorithm must play $x_t = a$ in each round in order to have at least $gain = a$.
- The adversary knows this, and will therefore play $u_t = b$
- Therefore, $gain_{Alg} = a$, $gain_{opt} = b$, competitive ratio $r = b/a$.

As usually, we ask whether randomization might help! Let's try the ski rental trick immediately! In particular, for all possible inputs $u \in [a, b]$ we want the same competitive ratio:

$$r \cdot gain_{Alg}(u) = gain_{opt}(u) = u.$$

From the deterministic case we know that it might make sense to treat the case $x = a$ individually. (If we do not, then the probability to choose $x = a$ will be infinitesimally small, and the adversary only needs to present $u = a + \epsilon$ all the time, and our algorithm is in trouble since it never makes any gain.)

We choose $x = a$ with probability p_a, and any value in $x \in (a, b]$ with probability density function $p(x)$, with $p_a + \int_a^b p(x)dx = 1$.

Theorem 7.32. *There is an algorithm that is r-competitive, with $r = 1 + \ln \frac{b}{a}$, "ln" being the natural logarithm.*

Proof. Setting up the ski rental trick, we have

$$r \cdot \left(p_a \cdot a + \int_a^u p(x) \cdot x dx \right) = u.$$

Then we differentiate with respect to u, and get,

$$\frac{\delta}{\delta u} = r \cdot p(u) \cdot u = 1 \Rightarrow p(u) = \frac{1}{ru}.$$

We plug this back into the differential equation, and get

$$r \cdot \left(p_a \cdot a + \int_a^u \frac{x}{rx} dx \right) = r p_a a + (u - a) = u \Rightarrow a(r p_a - 1) = 0 \Rightarrow p_a = 1/r.$$

To figure out r, we use that all probabilities must sum up to 1:

$$1 = p_a + \int_a^b p(x)dx = \frac{1}{r} + \frac{1}{r} \int_a^b \frac{1}{x} dx \Rightarrow 1 + \ln b - \ln a = r.$$

□

What about the lower bound? We use the Von Neumann / Yao Principle:

Theorem 7.33. *There is no randomized algorithm which is better than r-competitive, with $r = 1 + \ln \frac{b}{a}$.*

Proof. Let a little fairy tell us the right input distribution: We choose b with probability $p_b = a/b$, and select $u \in [a, b)$ with probability density $p(u) = a/u^2$. The input is OK because

$$p_b + \int_a^b \frac{a}{u^2} du = \frac{a}{b} + a \int_a^b \frac{1}{u^2} du = \frac{a}{b} + a \left(\frac{-1}{b} - \frac{-1}{a} \right) = 1.$$

The gain of the optimal algorithm on this input is:

$$gain_{opt} = b \cdot p_b + \int_a^b u \cdot p(u) du = b \frac{a}{b} + \int_a^b u \cdot \frac{a}{u^2} du = a + a \int_a^b \frac{1}{u} du = a(1 + \ln(b/a)).$$

The gain of a deterministic algorithm choosing x on this input is:

$$gain_x = x \cdot p_b + \int_x^b x \cdot p(u)du = x\frac{a}{b} + ax\int_x^b \frac{1}{u^2}du = ax(\frac{1}{b} + (\frac{-1}{b} - \frac{-1}{x})) = a.$$

Hence,

$$\frac{gain_{opt}}{gain_x} = \frac{a(1+\ln(b/a))}{a} = 1 + \ln(b/a).$$

□

Remarks:

- Great, upper and lower bound are tight!
- Didn't we ask for u, x being integers? In this case, $r = 1 + H_b - H_a$, where H_n is the harmonic number n defined as $H_n = \sum_{i=1}^{n} 1/i \approx \ln n$.
- Now let's turn to the more realistic cases where the bandwidth smoothly changes over time, and does not jump up and down like crazy.

7.11 Multiplicatively Changing Bandwidth

Now the adversary must choose u_t such that $u_t/\mu \leq u_{t+1} \leq \mu \cdot u_t$. The algorithm knows the maximal possible change factor μ per period. We assume that the algorithm also knows the initial threshold u_1. Think of μ as being a value such that the bandwidth changes a few percents only per period.

If the adversary keeps raising u as fast as possible ($u_{t+1} = \mu \cdot u_t$ for several rounds), then it seems reasonable that the algorithm does the same. In particular, if the algorithm chooses $x_{t+1} = (1-\epsilon)\mu x_t$ then

$$\lim_{t\to\infty} \frac{u_t}{x_t} = \frac{\mu^t}{(1-\epsilon)^t \cdot \mu^t} = \infty.$$

Therefore, if there was a successful transmission in period t, the algorithm chooses $x_{t+1} = \mu x_t$. On the other hand, if x_t was not successful, $x_{t+1} = \lambda x_t$. We will set $\lambda = 1/\mu^3$. The idea is that at least every other round is successful.

Lemma 7.34. *After a non-successful round there is always a successful round.*

Proof. Since we know u_1, the algorithm can choose $x_1 = u_1$, and have a success. Our invariant is that every non-successful round is followed by a successful round. Assume, for the sake of contradiction, that round $t+1$ is the first non-successful round which follows after a non-successful round t, which (by induction hypothesis) follows a successful round $t-1$ (note that $x_{t-1} \leq u_{t-1}$). Since $u_t \geq u_{t-1}/\mu$ for all t we have $u_{t+1} \geq u_{t-1}/\mu^2$. On the other hand, we have $x_{t+1} = \lambda x_t = \lambda\mu x_{t-1} = x_{t-1}/\mu^2$. Therefore,

$$x_{t+1} = x_{t-1}/\mu^2 \leq u_{t-1}/\mu^2 \leq u_{t+1},$$

hence round $t+1$ is a success. We have a contradiction, which proves that there can be only one non-successful round in a row. □

Lemma 7.35. *A successful round is μ^4-competitive.*

Proof.
- If a successful round $t+1$ follows a successful round t, round $t+1$ is at least as competitive as round t since the algorithm set $x_{t+1} = \mu x_t$.
- If a successful round $t+1$ follows a non-successful round t ($u_t < x_t$), then, since $x_{t+1} = \lambda x_t$ and $u_{t+1} \leq \mu u_t$ we have

$$x_{t+1} = \lambda x_t > \lambda u_t \geq \lambda u_{t+1}/\mu = u_{t+1}/\mu^4.$$

□

Theorem 7.36. *The algorithm is $(\mu^4 + \mu)$-competitive.*

Proof. In a non-successful ("fail") round t, it holds that $u_t < \mu x_{t-1}$, because $x_{t-1} \leq u_{t-1}$ (cf. Lemma 7.34), $x_t = \mu x_{t-1}$ and $u_t < \mu x_{t-1}$. Thus

$$\frac{gain_{opt}(succ) + gain_{opt}(fail)}{gain_{Alg}(succ)} < \frac{\mu^4 \cdot gain_{Alg}(succ) + \mu \cdot gain_{Alg}(succ)}{gain_{Alg}(succ)} = \mu^4 + \mu.$$

□

While this algorithm is good for small μ, the competitive ratio grows quickly for larger μ. In the following, we show that an algorithm which increases the bandwidth by a factor μ after successful rounds and halves the rate after non-successful rounds is 4μ-competitive.

Theorem 7.37. *This new algorithm is 4μ-competitive.*

Proof. First, we show by induction that in each successful or *good* round t, $u_t \leq 2\mu x_t$. For $t = 1$, $u_1 = x_1$ and the claim holds. For the induction step, consider the round $t-1$ before the good round t. There are two possibilities: either round $t-1$ was non-successful or *bad* ($x_{t-1} > u_{t-1}$), or good ($x_{t-1} \leq u_{t-1}$). If round $t-1$ was bad, we have $x_t = x_{t-1}/2$ and $u_t \leq u_{t-1}\mu < x_{t-1}\mu = 2\mu x_t$, hence $u_t/x_t < 2\mu$, and the claim holds. If on the other hand round $t-1$ was good, the algorithm increases the bandwidth at least as much as the adversary. Together with the induction hypothesis, the claim follows also in this case.

Having studied the gain in good rounds, we now consider bad rounds. We show that in the bad rounds following a good round t, the adversary may increase its gain at most by $2\mu x_t$. So let t be the good round preceding a sequence of bad rounds, t.e., $x_t \leq u_t$, $x_{t+1} > u_{t+1}$, $x_{t+2} > u_{t+2}$, etc. We know that $x_{t+1} = \mu x_t$, so—because it is a bad round—u_{t+1} must be less than μx_t. Further, we have $x_{t+2} = x_{t+1}/2 = \mu x_t/2$ and hence $u_{t+2} < \mu x_t/2$, $x_{t+3} = \mu x_t/4$ and hence $u_{t+3} < \mu x_t/8$, etc. By a geometric series argument, the gain of the adversary in the bad rounds is upper bounded by $2\mu x_t$.

Therefore,

$$\begin{aligned} \rho &= \frac{gain_{opt}(succ) + gain_{opt}(fail)}{gain_{Alg}(succ)} \\ &< \frac{2\mu \cdot gain_{Alg}(succ) + 2\mu \cdot gain_{Alg}(succ)}{gain_{Alg}(succ)} \\ &< 4\mu. \end{aligned}$$

□

7.12 Changes with Bursts

In the previous section, we assumed that the bandwidth changes by at most a given constant percentage μ over time. However, one can imagine that in the real Internet there may be quiet times where the congestion level hardly changes, and times where there are very abrupt or *bursty* changes. In main objective of this section is to present—without any analyses—an adversary model which incorporates such a notion of bursts. Our model is based on concepts of *network calculus*, a tool which is typically used to study queuing systems from a worst-case perspective.

The bursty adversary $\mathcal{ADV}_{nc}$ has two parameters: A *rate* $\mu \geq 1$ and *maximum burst factor* $B \geq 1$. In every round, the available bandwidth u_t may vary according to these parameters in a multiplicative manner. More precisely, $\mathcal{ADV}_{nc}$ may select the new bandwidth u_{t+1} from the interval

$$\mathcal{ADV}_{nc} : u_{t+1} \in [\frac{u_t}{\beta_t \mu}, u_t \cdot \beta_t \cdot \mu],$$

that is, the available bandwidth may change by a factor of at most $\beta_t \mu$. Thereby, β_t is the *burst factor at time t*. This burst factor is explained next.

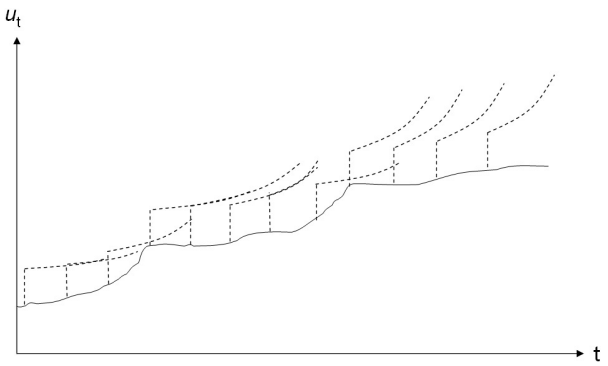

Figure 7.38: Visualization of $\mathcal{ADV}_{nc}$ for the case $\forall t : u_{t+1} \geq u_t$. The bandwidth may increase multiplicatively in every round, but it must never exceed the constraints from previous rounds (dashed lines).

On average, the available bandwidth can change by a factor μ per round. However, there may be times of only small changes, but then the bandwidth might change by factors larger than μ in later rounds. This is modeled with the burst factor β_t, which is defined as follows. At the beginning, β_t equals B, i.e., $\beta_1 = B$. For $t > 1$, the burst factor β_t is computed depending on β_{t-1} and the actual bandwidth change c_{t-1} that has happened in round $t-1$. More precisely,

$$\beta_t = \min\{B, \beta_{t-1} \frac{\mu}{c_{t-1}}\}$$

where $c_t := \frac{u_{t+1}}{u_t}$ if $u_{t+1} > u_t$ and $\frac{u_t}{u_{t+1}}$ otherwise. This means that if the available bandwidth changed by a factor less than μ in round t, i.e., $c_t < \mu$, the burst factor *increased* by a factor $\frac{\mu}{c_t}$, and hence the bandwidth can change more in the next round, and vice versa if $c_t > \mu$.

Therefore, the adversary is allowed to save adversarial power for forthcoming rounds. However, this amortization is limited as β_t can never become larger than B for all rounds t. Also note that $\beta_t \geq 1$ always holds, because $c_t \leq \mu\beta_t$ by the definition of $\mathcal{ADV}_{nc}$.

Figure 7.38 visualizes $\mathcal{ADV}_{nc}$ for the case $\forall t : u_{t+1} \geq u_t$, i.e., for increasing bandwidth only: The bandwidth may rise by a factor of μB in every round, unless it conflicts with a constraint from a previous round, i.e., $\forall t : u_t \leq \min_{i \in \{1,\ldots,t-1\}} \{u_t \cdot B \cdot \mu^{t-i}\}$.

In order to analyze such bursty adversaries, similar techniques as those presented in Section 7.11 can be applied; we do not perform these computations here.