(4.10) *The schedule A produced by the greedy algorithm has optimal maximum lateness L.*

Proof. Statement (4.9) proves that an optimal schedule with no inversions exists. Now by (4.8) all schedules with no inversions have the same maximum lateness, and so the schedule obtained by the greedy algorithm is optimal. ■

Extensions

There are many possible generalizations of this scheduling problem. For example, we assumed that all jobs were available to start at the common start time s. A natural, but harder, version of this problem would contain requests i that, in addition to the deadline d_i and the requested time t_i, would also have an earliest possible starting time r_i. This earliest possible starting time is usually referred to as the *release time*. Problems with release times arise naturally in scheduling problems where requests can take the form: Can I reserve the room for a two-hour lecture, sometime between 1 P.M. and 5 P.M.? Our proof that the greedy algorithm finds an optimal solution relied crucially on the fact that all jobs were available at the common start time s. (Do you see where?) Unfortunately, as we will see later in the book, in Chapter 8, this more general version of the problem is much more difficult to solve optimally.

4.3 Optimal Caching: A More Complex Exchange Argument

We now consider a problem that involves processing a sequence of requests of a different form, and we develop an algorithm whose analysis requires a more subtle use of the exchange argument. The problem is that of *cache maintenance*.

The Problem

To motivate caching, consider the following situation. You're working on a long research paper, and your draconian library will only allow you to have eight books checked out at once. You know that you'll probably need more than this over the course of working on the paper, but at any point in time, you'd like to have ready access to the eight books that are most relevant at that time. How should you decide which books to check out, and when should you return some in exchange for others, to minimize the number of times you have to exchange a book at the library?

This is precisely the problem that arises when dealing with a *memory hierarchy*: There is a small amount of data that can be accessed very quickly,

and a large amount of data that requires more time to access; and you must decide which pieces of data to have close at hand.

Memory hierarchies have been a ubiquitous feature of computers since very early in their history. To begin with, data in the main memory of a processor can be accessed much more quickly than the data on its hard disk; but the disk has much more storage capacity. Thus, it is important to keep the most regularly used pieces of data in main memory, and go to disk as infrequently as possible. The same phenomenon, qualitatively, occurs with on-chip caches in modern processors. These can be accessed in a few cycles, and so data can be retrieved from cache much more quickly than it can be retrieved from main memory. This is another level of hierarchy: small caches have faster access time than main memory, which in turn is smaller and faster to access than disk. And one can see extensions of this hierarchy in many other settings. When one uses a Web browser, the disk often acts as a cache for frequently visited Web pages, since going to disk is still much faster than downloading something over the Internet.

Caching is a general term for the process of storing a small amount of data in a fast memory so as to reduce the amount of time spent interacting with a slow memory. In the previous examples, the on-chip cache reduces the need to fetch data from main memory, the main memory acts as a cache for the disk, and the disk acts as a cache for the Internet. (Much as your desk acts as a cache for the campus library, and the assorted facts you're able to remember without looking them up constitute a cache for the books on your desk.)

For caching to be as effective as possible, it should generally be the case that when you go to access a piece of data, it is already in the cache. To achieve this, a *cache maintenance* algorithm determines what to keep in the cache and what to evict from the cache when new data needs to be brought in.

Of course, as the caching problem arises in different settings, it involves various different considerations based on the underlying technology. For our purposes here, though, we take an abstract view of the problem that underlies most of these settings. We consider a set U of n pieces of data stored in *main memory*. We also have a faster memory, the *cache*, that can hold $k < n$ pieces of data at any one time. We will assume that the cache initially holds some set of k items. A sequence of data items $D = d_1, d_2, \ldots, d_m$ drawn from U is presented to us—this is the sequence of memory references we must process—and in processing them we must decide at all times which k items to keep in the cache. When item d_i is presented, we can access it very quickly if it is already in the cache; otherwise, we are required to bring it from main memory into the cache and, if the cache is full, to *evict* some other piece of data that is currently in the cache to make room for d_i. This is called a *cache miss*, and we want to have as few of these as possible.

Thus, on a particular sequence of memory references, a cache maintenance algorithm determines an *eviction schedule*—specifying which items should be evicted from the cache at which points in the sequence—and this determines the contents of the cache and the number of misses over time. Let's consider an example of this process.

- Suppose we have three items $\{a, b, c\}$, the cache size is $k = 2$, and we are presented with the sequence

$$a, b, c, b, c, a, b.$$

  Suppose that the cache initially contains the items a and b. Then on the third item in the sequence, we could evict a so as to bring in c; and on the sixth item we could evict c so as to bring in a; we thereby incur two cache misses over the whole sequence. After thinking about it, one concludes that any eviction schedule for this sequence must include at least two cache misses.

Under real operating conditions, cache maintenance algorithms must process memory references $d_1, d_2, \ldots$ without knowledge of what's coming in the future; but for purposes of evaluating the quality of these algorithms, systems researchers very early on sought to understand the nature of the optimal solution to the caching problem. Given a full sequence S of memory references, what is the eviction schedule that incurs as few cache misses as possible?

Designing and Analyzing the Algorithm

In the 1960s, Les Belady showed that the following simple rule will always incur the minimum number of misses:

```
When d_i needs to be brought into the cache,
  evict the item that is needed the farthest into the future
```

We will call this the *Farthest-in-Future Algorithm*. When it is time to evict something, we look at the next time that each item in the cache will be referenced, and choose the one for which this is as late as possible.

This is a very natural algorithm. At the same time, the fact that it is optimal on all sequences is somewhat more subtle than it first appears. Why evict the item that is needed farthest in the future, as opposed, for example, to the one that will be used least frequently in the future? Moreover, consider a sequence like

$$a, b, c, d, a, d, e, a, d, b, c$$

with $k = 3$ and items $\{a, b, c\}$ initially in the cache. The Farthest-in-Future rule will produce a schedule S that evicts c on the fourth step and b on the seventh step. But there are other eviction schedules that are just as good. Consider the schedule S' that evicts b on the fourth step and c on the seventh step, incurring the same number of misses. So in fact it's easy to find cases where schedules produced by rules other than Farthest-in-Future are also optimal; and given this flexibility, why might a deviation from Farthest-in-Future early on not yield an actual savings farther along in the sequence? For example, on the seventh step in our example, the schedule S' is actually evicting an item (c) that is needed *farther* into the future than the item evicted at this point by Farthest-in-Future, since Farthest-in-Future gave up c earlier on.

These are some of the kinds of things one should worry about before concluding that Farthest-in-Future really is optimal. In thinking about the example above, we quickly appreciate that it doesn't really matter whether b or c is evicted at the fourth step, since the other one should be evicted at the seventh step; so given a schedule where b is evicted first, we can swap the choices of b and c without changing the cost. This reasoning—swapping one decision for another—forms the first outline of an *exchange argument* that proves the optimality of Farthest-in-Future.

Before delving into this analysis, let's clear up one important issue. All the cache maintenance algorithms we've been considering so far produce schedules that only bring an item d into the cache in a step i if there is a request to d in step i, and d is not already in the cache. Let us call such a schedule *reduced*—it does the minimal amount of work necessary in a given step. But in general one could imagine an algorithm that produced schedules that are not reduced, by bringing in items in steps when they are not requested. We now show that for every nonreduced schedule, there is an equally good reduced schedule.

Let S be a schedule that may not be reduced. We define a new schedule $\overline{S}$—the *reduction* of S—as follows. In any step i where S brings in an item d that has not been requested, our construction of $\overline{S}$ "pretends" to do this but actually leaves d in main memory. It only really brings d into the cache in the next step j after this in which d is requested. In this way, the cache miss incurred by $\overline{S}$ in step j can be charged to the earlier cache operation performed by S in step i, when it brought in d. Hence we have the following fact.

(4.11) *$\overline{S}$ is a reduced schedule that brings in at most as many items as the schedule S.*

Note that for any reduced schedule, the number of items that are brought in is exactly the number of misses.

Proving the Optimalthy of Farthest-in-Future We now proceed with the exchange argument showing that Farthest-in-Future is optimal. Consider an arbitrary sequence D of memory references; let S_{FF} denote the schedule produced by Farthest-in-Future, and let S^* denote a schedule that incurs the minimum possible number of misses. We will now gradually "transform" the schedule S^* into the schedule S_{FF}, one eviction decision at a time, without increasing the number of misses.

Here is the basic fact we use to perform one step in the transformation.

(4.12) *Let S be a reduced schedule that makes the same eviction decisions as S_{FF} through the first j items in the sequence, for a number j. Then there is a reduced schedule S' that makes the same eviction decisions as S_{FF} through the first $j+1$ items, and incurs no more misses than S does.*

Proof. Consider the $(j+1)^{\text{st}}$ request, to item $d = d_{j+1}$. Since S and S_{FF} have agreed up to this point, they have the same cache contents. If d is in the cache for both, then no eviction decision is necessary (both schedules are reduced), and so S in fact agrees with S_{FF} through step $j+1$, and we can set $S' = S$. Similarly, if d needs to be brought into the cache, but S and S_{FF} both evict the same item to make room for d, then we can again set $S' = S$.

So the interesting case arises when d needs to be brought into the cache, and to do this S evicts item f while S_{FF} evicts item $e \neq f$. Here S and S_{FF} do not already agree through step $j+1$ since S has e in cache while S_{FF} has f in cache. Hence we must actually do something nontrivial to construct S'.

As a first step, we should have S' evict e rather than f. Now we need to further ensure that S' incurs no more misses than S. An easy way to do this would be to have S' agree with S for the remainder of the sequence; but this is no longer possible, since S and S' have slightly different caches from this point onward. So instead we'll have S' try to get its cache back to the same state as S as quickly as possible, while not incurring unnecessary misses. Once the caches are the same, we can finish the construction of S' by just having it behave like S.

Specifically, from request $j+2$ onward, S' behaves exactly like S until one of the following things happens for the first time.

(i) There is a request to an item $g \neq e, f$ that is not in the cache of S, and S evicts e to make room for it. Since S' and S only differ on e and f, it must be that g is not in the cache of S' either; so we can have S' evict f, and now the caches of S and S' are the same. We can then have S' behave exactly like S for the rest of the sequence.

(ii) There is a request to f, and S evicts an item e'. If $e' = e$, then we're all set: S' can simply access f from the cache, and after this step the caches

of S and S' will be the same. If $e' \neq e$, then we have S' evict e' as well, and bring in e from main memory; this too results in S and S' having the same caches. However, we must be careful here, since S' is no longer a reduced schedule: it brought in e when it wasn't immediately needed. So to finish this part of the construction, we further transform S' to its reduction $\overline{S'}$ using (4.11); this doesn't increase the number of items brought in by S', and it still agrees with S_{FF} through step $j+1$.

Hence, in both these cases, we have a new reduced schedule S' that agrees with S_{FF} through the first $j+1$ items and incurs no more misses than S does. And crucially—here is where we use the defining property of the Farthest-in-Future Algorithm—one of these two cases will arise *before* there is a reference to e. This is because in step $j+1$, Farthest-in-Future evicted the item (e) that would be needed farthest in the future; so before there could be a request to e, there would have to be a request to f, and then case (ii) above would apply. ■

Using this result, it is easy to complete the proof of optimality. We begin with an optimal schedule S^*, and use (4.12) to construct a schedule S_1 that agrees with S_{FF} through the first step. We continue applying (4.12) inductively for $j = 1, 2, 3, \ldots, m$, producing schedules S_j that agree with S_{FF} through the first j steps. Each schedule incurs no more misses than the previous one; and by definition $S_m = S_{FF}$, since it agrees with it through the whole sequence. Thus we have

(4.13) *S_{FF} incurs no more misses than any other schedule S^* and hence is optimal.*

Extensions: Caching under Real Operating Conditions

As mentioned in the previous subsection, Belady's optimal algorithm provides a benchmark for caching performance; but in applications, one generally must make eviction decisions on the fly without knowledge of future requests. Experimentally, the best caching algorithms under this requirement seem to be variants of the *Least-Recently-Used* (LRU) Principle, which proposes evicting the item from the cache that was referenced *longest ago*.

If one thinks about it, this is just Belady's Algorithm with the direction of time reversed—longest in the past rather than farthest in the future. It is effective because applications generally exhibit *locality of reference*: a running program will generally keep accessing the things it has just been accessing. (It is easy to invent pathological exceptions to this principle, but these are relatively rare in practice.) Thus one wants to keep the more recently referenced items in the cache.

Long after the adoption of LRU in practice, Sleator and Tarjan showed that one could actually provide some theoretical analysis of the performance of LRU, bounding the number of misses it incurs relative to Farthest-in-Future. We will discuss this analysis, as well as the analysis of a randomized variant on LRU, when we return to the caching problem in Chapter 13.

4.4 Shortest Paths in a Graph

Some of the basic algorithms for graphs are based on greedy design principles. Here we apply a greedy algorithm to the problem of finding shortest paths, and in the next section we look at the construction of minimum-cost spanning trees.

The Problem

As we've seen, graphs are often used to model networks in which one travels from one point to another—traversing a sequence of highways through interchanges, or traversing a sequence of communication links through intermediate routers. As a result, a basic algorithmic problem is to determine the shortest path between nodes in a graph. We may ask this as a point-to-point question: Given nodes u and v, what is the shortest u-v path? Or we may ask for more information: Given a *start node* s, what is the shortest path from s to each other node?

The concrete setup of the shortest paths problem is as follows. We are given a directed graph $G = (V, E)$, with a designated start node s. We assume that s has a path to every other node in G. Each edge e has a length $\ell_e \geq 0$, indicating the time (or distance, or cost) it takes to traverse e. For a path P, the *length of* P—denoted $\ell(P)$—is the sum of the lengths of all edges in P. Our goal is to determine the shortest path from s to every other node in the graph. We should mention that although the problem is specified for a directed graph, we can handle the case of an undirected graph by simply replacing each undirected edge $e = (u, v)$ of length ℓ_e by two directed edges (u, v) and (v, u), each of length ℓ_e.

Designing the Algorithm

In 1959, Edsger Dijkstra proposed a very simple greedy algorithm to solve the single-source shortest-paths problem. We begin by describing an algorithm that just determines the *length* of the shortest path from s to each other node in the graph; it is then easy to produce the paths as well. The algorithm maintains a set S of vertices u for which we have determined a shortest-path distance $d(u)$ from s; this is the "explored" part of the graph. Initially $S = \{s\}$, and $d(s) = 0$. Now, for each node $v \in V - S$, we determine the shortest path that can be constructed by traveling along a path through the explored part S to some $u \in S$, followed by the single edge (u, v). That is, we consider the quantity