Greedy algorithms: Minimising lateness and optimal caching

1 The problem set up: Minimising lateness

We go back to the situation when we possess one resource. The resource is available to be used by requests starting at time s. We again have several requests: r_1, r_2, ..., r_n. This time the requests are more flexible in the sense that each r_i indicates the length of time t_i needed to continuously occupy the resource. But, this flexibility is constrained with each request r_i having its deadline d_i. We would like to schedule requests such that the following two goals are met:

1. The requests do not overlap.
2. The deadlines are satisfied to the best of our capacity.

Clearly, we need to clarify each of these two goals. The first tells us that we need to schedule all requests by allocating starting time $s(i)$ and finishing time $f(i)$ for r_i such that the length of the interval equals t_i. Clearly, it suffices to select a starting time $s(i)$ since the finishing time must be equal to $s(i) + t_i$. Moreover, the allocated intervals

$$[s(1), f(1)], [s(2), f(2)], \ldots, [s(n), f(n)]$$

must form a compatible set of intervals. We now need to formally explain the second goal. Suppose we have allocated intervals to the requests such that the set of allocated intervals forms a compatible set. Consider the interval $[s(i), f(i)]$ allocated to the request r_i. We say that the request r_i is late if $f(i) > d_i$. When r_i is late, the value

$$\ell_i = f(i) - d_i$$

indicates *lateness* of the request to meet the deadline d_i. If when $f(i) - d_i < 0$, then we say that request r_i meets the deadline (so r_i is not late), and we indicate this fact by setting $\ell_i = 0$. Consider the *maximum possible lateness value* M with respect to the given schedule:

$$M = \max\{\ell_1, \ell_2, \ldots, \ell_n\}.$$

The goal is to schedule the requests in such a way that M reaches its minimum possible value. In other words, we want to minimise the maximum value of lateness among *all possible schedules*. Let us call this problem *Minimising lateness* problem. Any scheduling that provides the minimum possible value for M is called *optimal* scheduling.

Exercise 1. Explain why the minimising lateness problem has an optimal solution.

2 Designing the algorithm

We represent requests r_1, ..., r_n as the list: (t_1, d_1), (t_2, d_2), ..., (t_n, d_n), where each t_i is the length of time r_i wants to use the resource and d_i is the deadline. What would a greedy strategy be in order to minimise lateness? There are several approaches:

1. *Schedule those requests that occupy the resources the shortest amount of time.* In other words, the idea is to sort the sequence $t_1, \ldots, t_n$, and schedule the requests according to the sorted list. This is a quite natural approach since we want to schedule short requests as quickly as possible. Intuitively, the problem with this approach is that it ignores deadlines. In other words, half of the input is ignored.
2. *Schedule those requests that have the least slack time.* A slack time of request r_i is the difference $d_i - t_i$. This approach clearly takes into account the deadlines. The problem with this approach is that requests with vastly different requests times for occupying the resource might have the same amount of slack time.

Exercise 2. Give examples of schedules where the two rules described above do not produce optimal solution.

Here is another greedy rule. *Schedule those requests that have earliest deadlines.* Let us call this rule *Earliest Deadline First* rule. This is a natural rule as we want to make sure that requests with the earlier deadlines complete their jobs first. One might immediately argue that, the first rule above that we considered ignored deadlines, and that was the reason for us to be skeptical about the rule. The *Earliest Deadline First* approach also ignores one half of the input, namely the lengths t_i. So, why would this approach work? The answer to the question is that the "ignored" half of the input

$$t_1, \ldots, t_n$$

is actually used in our algorithm and proof and that this new rule produces an optimal solution.

So, we sort the requests according to their deadlines. Let us assume, if need be by renaming the requests, the deadlines are ordered as follows:

$$d_1 \leq d_2 \leq \ldots \leq d_i \leq \ldots \leq d_n.$$

Recall that we have a starting time s from which our resource can be used. We schedule the requests in this order of deadlines. Request r_1 occupies the resource from $s(1) = s$ till $f(1) = s(1) + t_1$, request r_2 occupies the resource from time $s(2) = f(1)$ to $f(2) = s(2) + t_2$, and so on. Here is the algorithm:

Algorithm 1 Earliest Deadline First Algorithm

Order requests according to their deadline
Let $d_1 \leq \ldots \leq d_n$ be the order
Set $f = s$
− For request r_i, where $i = 1, \ldots, n$
−− Assign r_i the time interval from $s(i) = f$ to $f(i) = f + t_i$
−− Let $f = f + t_i$
− EndFor
Return the set of scheduled intervals $[s(i), f(i)]$ for $i = 1, \ldots, n$.

Now our goal is to show that the algorithm above is a correct algorithm; that is, it provides an optimal solution to the minimising lateness problem.

3 Analysing the algorithm

A simple observation is concerned with *idle times* of the resource (machine). Suppose that there are requests left that need to occupy the machine but the machine sits idle and not working. Our algorithm above avoids these type of idle times. The machine is busy continuously. In fact, the following is quite easy to see:

Observation 1 *Every optimal schedule has no idle time.*

Exercise 3. Give a formal explanation that the observation is correct.

Let A be the schedule produced by our algorithm. We compute its lateness, that is, let L be the maxmmum value among all ℓ_1, ℓ_2, ..., ℓ_n the lateness values with respect to A. Our goal is to show that the value L is as minimal as possible. For this we need to explain why the maximum lateness value of any optimal schedule equals L.

Let us now consider an optimal schedule $\mathcal{O}$ that solves the problem. We would like to connect our schedule A with $\mathcal{O}$ with the hope of showing that A is optimal.

What does connect mean in this setting? The idea is to modify $\mathcal{O}$ step-by-step without effecting its optimality. This step-by-step transformation of $\mathcal{O}$ will eventually produce our schedule A. Since, at each step optimality is preserved, this would show that A is an optimal schedule. This type of reasoning is known as an *exchange argument*.

The next observation easily follows from our discussions above and the algorithm:

Observation 2 *The schedule A produced by our algorithm has the following two nice properties:*

1. *If request r_i has earlier deadline than request r_j then r_i is scheduled earlier that r_j.*
2. *The schedule A has no idle time.*

Definition 1. *Call schedules that satisfy the two properties of the proposition* **good** *schedules.*

Note that if all the deadlines are pairwise distinct (that is $d_i \neq d_j$ for all $i \neq j$), then there is only one good schedule. However, if there are k equal deadlines then there are $k!$ *good* schedules. An interesting observation is this:

Observation 3 *All good schedules have the same maximum lateness.*

Proof. Two good schedules, say S_1 and S_2, can only differ in scheduling those requests with the same deadline. So, say d is the least deadline at which the two schedules differ in scheduling requests. So, both S_1 and S_2 have the following properties:

1. In both S_1 and S_2 scheduling the requests with deadline d starts at the same time.
2. In both S_1 and S_2 the finishing times of all requests with deadline d are the same.
3. In both S_1 and S_2 scheduling requests with deadlines greater than d starts at the same time.

Since the finishing times of all requests with deadline d, in both S_1 and S_2, are the same we have the following. Among all the requests with deadline d, the last scheduled request has the greatest lateness. This lateness does *not* depend on the order of these request. This argument now implies that S_1 and S_2 achieve the same maximum lateness.

Now, having the above observation, we have an idea how to show that the schedule A constructed by the algorithm is optimal: We need to prove that there is a good schedule which is optimal.

So, let us take an optimal solution $\mathcal{O}$. We can assume that $\mathcal{O}$ has no idle time. If $\mathcal{O}$ is good then we are done. Otherwise, there are two requests p and q with deadlines d_p and d_q such that $d_p < d_q$ but $\mathcal{O}$ schedules q earlier than p. We call such pairs (p, q) *inversions* of the schedule $\mathcal{O}$. In case p is scheduled right after q then we say that the inversion is an *immediate inversion.*

Observation 4 *If $\mathcal{O}$ has an inversion then it has an immediate inversion.*

Indeed, take an inversion (p, q). Now let us move, according to the schedule $\mathcal{O}$, from request p to request q one by one. There must clearly be a request a such that for the next request b we have $d_b < d_a$. The pair (a, b) is an immediate inversion pair. This proves the observation.

So, for the optimal solution $\mathcal{O}$ we now proceed as follows. Take an immediate inversion pair (p, q) in $\mathcal{O}$ which we now know that exists. Let us swap p and q in the schedule. This produces a new schedule $\mathcal{O}'$. Here is a simple observation whose proof is left to the reader:

Observation 5 *The new schedule $\mathcal{O}'$ has one less inversions than the schedule $\mathcal{O}$.*

Exercise 4. Prove the observation.

The statement that needs a good explanation is the next observation:

Observation 6 *The schedule $\mathcal{O}'$ is still optimal.*

The proof of this observation is a beautiful argument. We need some easy notations, and some clear notes about these notations.

- Let s_1 be the time when all requests according to $\mathcal{O}$ that are scheduled before p finish their jobs. In other words, s_1 is the time when p starts using the resource according to $\mathcal{O}$.
- Let f_1 be the time, according to $\mathcal{O}$, when p finishes the job.
- Let s_2 be the time, according to $\mathcal{O}$, when q starts using the resource. Clearly $s_2 = f_1$.
- Let f_2 be the time, according to $\mathcal{O}$, when q finishes the job.
- Let s'_1 be the time when all requests according to $\mathcal{O}'$ that are scheduled before q finish their jobs. In other words, s'_1 is the time when q starts using the resource according to $\mathcal{O}'$.
- Let f'_1 be the time, according to $\mathcal{O}'$, when q finishes the job.
- Let s'_2 be the time, according to $\mathcal{O}'$, when p starts using the resource. Clearly $s'_2 = f'_1$.
- Let f'_2 be the time, according to $\mathcal{O}'$, when p finishes the job.

It is obvious that $s_1 = s'_1$ and $f_2 = f'_2$. This implies that the latenesses of all the requests, apart from p and q, in both schedules $\mathcal{O}$ and $\mathcal{O}'$ are the same. Note also that the lateness of request p in $\mathcal{O}'$ is smaller than the lateness of p in $\mathcal{O}$; this is simply because p starts using the resource earlier in $\mathcal{O}'$ than in $\mathcal{O}$. The only value that we need to worry about is l'_q the lateness of q in $\mathcal{O}'$. Note that $l'_q = f_2 - d_q$. Also, note that the lateness of q in $\mathcal{O}$ is calculated as $l_q = f_2 - d_q$. Now, we use the fact that $d_q < d_p$:

$$l'_q = f_2 - d_q \leq f_2 - d_p = l_p.$$

All these arguments imply that the maximum lateness of $\mathcal{O}'$ can not be greater than the maximum lateness of $\mathcal{O}$. Hence, the schedule $\mathcal{O}'$ is optimal. This finishes the explanation of the observation.

Now set the schedule $\mathcal{O}$ to be $\mathcal{O}'$ and remove another immediate inversion if it exists. Iterate this until no inversions are left. The resulting schedule $\mathcal{O}$ will be a good schedule. It will be an optimal schedule since swapping inversions preserves optimality. Hence, by Observation 3, the output A and $\mathcal{O}$ have the same maximum lateness. Since $\mathcal{O}$ is optimal, we conclude that the Algorithm 1 outputs an optimal solution.

What remains is the running time analysis. The assigning intervals to requests r_i is a constant time operation. So, it takes $O(n)$ time to schedule the requests given we have ordered the deadlines. Ordering the deadlines is $O(n \log(n))$ process. Thus, the algorithm runs in $O(n \log(n))$.

4 The problem set-up: Optimal Caching

Managing memory is an important topic. We know that data in the main memory of a computer system can be accessed much faster than data in hard disks. The balance, however, is achieved by the fact that the main memory holds much less amount of data as opposed to hard disks.

Caching is a process of storing a small and important amount of data in a memory that can be accessed quickly. The goal of caching is to reduce the amount of time that requires the interaction with slow memory. For instance, the main memory of a computer system plays the role of cache for the hard disk.

In general, memory of a system might consists of several levels:

$$M_1,\ M_2,\ \ldots,\ M_k.$$

All these store data. M_1 is a data storage that can be accessed the most quickly. M_1 acts as a cache for M_2. In turn, data in M_2 can be accessed much quicker than the data in M_3. So, M_2 acts as a cache for M_3, end so on. For instance, modern processors have on-chip caches. On-chip caches are accessed much faster than the data in the main memory. On-chip cache acts as a cache for the main memory, and the main memory acts as a cache for the hard disk. So, this already gives us three levels of memory: on-chip cache, main memory, and hard disk.

Here we simplify the situation and consider the following. We assume that the data is stored in the fast memory C, and also, in the main but slower memory M. So, the fast memory C acts as a cache for the main memory M. We assume that C can hold k pieces of data, and assume

C already possesses k data: c_1, ..., c_k. The storage size of the main memory M equals m and that m is much larger than k (since C acts as a cache for M).

We would like caching to be as effective as possible. Effectiveness is usually expressed in the following property:

When we access a piece of data, the data should already be in the cache.

To achieve this, there are many cache maintenance algorithms. Their goal is to determine what data to keep in the cache and what data needs to be evicted. Of course, these algorithms are dependent on the memory structure, the technology that is being used, and many other factors.

Here we address the cache optimisation problem in its pure and abstract setting. This setting still underlines many practical problems that arise in maintaining cache efficiently. Suppose a system is given a reference-key to data item d. If d is in the cache C, then the system accesses d quickly. If d is *not* in the cache, then this is called a *cache miss*. Recall that by our assumption the cache C is full. So, if d results in a cache miss, then this triggers the following action. The system accesses the memory M, retrieves d, removes an item x from C to make a room for d, puts x into M, and puts d into C (thus replacing x with d).

Now we state the *Optimal Caching Problem*. Assume that a sequence of data items

$$d_1, d_2, \ldots, d_n$$

is presented (as an input) to the system with cache C and main memory M. More precisely, these are all data reference keys that the system needs to process. The system reads this input one at a time. When it reads d_i, if d_i is in C then d_i is accessed quickly; otherwise, we have a cache miss. Our goal is to minimise the number of cache misses. Thus, we need to design an algorithm that given an input d_1, d_2, ..., d_n schedules the removal and replacement mechanism from the cache so that the number of evictions from the cache is as few as possible.

In practice, when we maintain cache we process the memory references $d_1, d_2, \ldots$ *without knowing* what comes next. Our formulation of the optimal cache problem assumes that we know the full input sequence. This is a disparity between practical cache maintenance problem and its theoretical formulation. However, in order to understand the cache maintenance problem in practice, it is important to investigate the problem in pure idealised setting. This is exactly what systems researchers did very early on.

5 Designing optimal caching algorithm

We start with a simple example.

Example 1. Suppose that M contains 4 items a, b, c, and d. Assume that cache C can hold just 2 items, and currently C holds a and d. Consider the following input:

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

On the first item b of the input, we need to evict either a or d from cache. Evicting a would clearly be a wrong choice if we want to minimise the number of cache misses. Note that d never

occurs in the input sequence; therefore the safest choice is to evict d from cache. So, after reading the first item b, our cache contains a and b. On the fourth item in the input sequence, we can evict either a or b. Intuitively, a is a good choice to evict since a occurs only once in the rest of the input and b occurs twice. So we evict a and put c into the cache. So, cache has b and c in it. On the eighth item we evict c as oppose to evicting b since b occurs in the rest of the sequence and c does not. So we put a back into cache. Thus, we calculated 3 cache misses. With some thought it is easy to see that 3 cache misses is the optimal amount for the input.

In 1960 Les Belady invented the following simply stated algorithm that solves the optimal caching problem:

Algorithm 2 Farthest-in-Future Algorithm

Let C be the cache which is full
On input d_1, d_2, …, d_n
When d_i should be put into cache C
Evict the item in C that is needed the farthest into the future.

So, given an input d_1, …, d_i, …, d_n, the algorithm proceeds as follows. When the algorithm considers item d_i, it checks if d_i causes a cash miss. If d_i causes a cache miss, it calculates which of the items in the current cache C would be used latest. Then the algorithm evicts that item, and brings in item d_i into cache. Thus, the algorithm on the input sequence d_1, …, d_n produces a scheduling S that removes items from the cache C.

As in our previous examples of greedy algorithms, this algorithm is simple to describe and it looks quite natural. The beauty of this algorithm is that it is correct: it always outputs an optimal eviction schedule. So, there are several questions. Why is the algorithm correct? Why would not evict the elements from cache that will be used less frequently? We address correctness of the algorithm in the next section. But for the second question solve the following exercise:

Exercise 5. Suppose the rule for eviction from the cache is the following. Remove the item from C that occurs less frequently in the input. Will this rule produce an optimal solution?

6 Why is Farthest-in-Future Algorithm correct?

Let S_{FF} be the schedule obtained by applying our Farthest-in-Future Algorithm to the input d_1, …, d_n. Our goal is to show that this schedule is optimal. Before we start analysing the schedule we would like to make one interesting point.

Our Farthest-in-Future Algorithm has the following property. Suppose that the input is d_1, …, d_n. When the algorithm considers item d_i from the input, it evicts an item from the cache when *necessary*. In other words, if d_i is in the cache then nothing is evicted from the cache, and the algorithm proceeds to the next item. Otherwise, an item is evicted from the cache. We call

such scheduling sequences *reduced sequences*. It is important to note that cache maintenance algorithms do *not* have to schedule reduced sequences. Some algorithms might evict items from the cache even if it is not necessary to do so. Nevertheless, we claim the following.

Claim. Every scheduling sequence S on input $d_1, \ldots, d_n$ can be replaced with a reduced scheduling sequence S_r on $d_1, \ldots, d_n$ such that S_r has at most as many cache misses as S does.

We outline an idea how to build S_r (as claimed) but note that showing that the idea works is a tedious exercise. Assume that S makes unnecessary eviction on reading item d_i. This means that d_i is in the cache, yet by reading d_i, the schedule S evicts an item c from C and replaces it with an item, say d. The new schedule S_r can intuitively be described as follows. The schedule S_r follows S up to the item d_i. Once d_i is seen, S_r "pretends" that it evicted c from C and replaced it with d, but S_r leaves c in C and d in M. After that, S_r continues on simulating S, and it only brings in d into the cache when d is requested. We note this argument is not precise but it is a good exercise to prove that the claim is true.

So, consider S_{FF} the schedule obtained by applying our Farthest-in-Future Algorithm to the input $d_1, \ldots, d_n$. We want to prove the following fact:

Fact 1 *Let S be a reduced schedule that acts the same way as S_{FF} through the first j items in the input sequence $d_1, \ldots, d_j, d_{j+1}, \ldots, d_n$. Then there is a reduced schedule S' that acts the same way as S_{FF} through the first $j+1$ of the input sequence, and has no more misses than S.*

Note that this fact implies that the algorithm is correct. Indeed, for $j = 0$ we take any reduced optimal schedule, lest call S_0, for the input sequence $d_1, \ldots, d_n$. By the fact above, there exists an optimal reduced schedule S_1 that agrees with S_{FF} at d_1. Now we apply the fact $n-1$ times, and obtain the sequence of reduced schedules $S_0, S_1, \ldots, S_n$. The last schedule S_n coincides with S_{FF}. The first schedule S_0 is our initial optimal schedule. By the fact above each S_{i+1} incurs no more misses than S_j. Hence, this implies that S_{FF} is an optimal schedule. This argument proves that the Farthest-in-Future-First Algorithm is a correct one. So, we are left to prove the fact.

Proof of the fact. Consider $d = d_{j+1}$. By assumption S and S_{FF} have the same cache at this time. If the cache has d, then we are done as we can set $S' = S$. If d is not in the cache, and both S and S_{FF} evict the same element from the cache, then we are done by setting $S' = S$.

Thus, the case to consider is when d needs to be put into the cache, S evicts item f from its cache, S_{FF} removes item e, and $e \neq f$. So, S and S_{FF} do not agree at step $j+1$ in the sense that S has e in its cache but S_{FF} does not, and S_{FF} has f in its cache but S does not.

Since we want to construct S' in such a way that it acts just as S_{FF} throughout the first $j+1$ items of the input, we let S' to act the same way as S_{FF} on the sequence $d_1, \ldots, d_j, d_{j+1}$. In particular, at step $j+1$, S' evicts e from cache and brings in d. Hence, the cache of S' is the same as the cache of S_{FF}. Therefore, at the end of step $j+1$, S has e in its cache but S' does not, and S' has f in its cache but S does not. Now the goal is to ensure, as quickly as possible, that the cache of S' becomes the same as the cache of S. From that point on S' will simply copy S. We also need to ensure that S' has no more cache misses than S. How do we guarantee this?

Well, from step $j+2$ on S' follows S until one of the following two cases occurs:

1. There is a request to an item g such that $g \neq e$, $g \neq f$, and S evicts e from its cache (to make a room for g).
2. There is a request to f, and S evicts an item e' to make a room for f.

Note that when any of these two cases occur, the caches of S and S' only differ in e and f. Suppose that Case 1 occurs, then g belongs to none of these cashes. Hence, S' just evicts f and puts in g into its cache. From this point on S and S' have the same cache. Hence, we have the desired S'. Suppose that Case 2 occurs. Then if $e' = e$, then S' has to do nothing since the schedule S by evicting e ensured that S and S' have the same cache. Assume that S evicted e' such that $e' \neq e$. Then S' makes the following move: it evicts e' and puts e into its cache. This guarantees that S and S' have the same cache. The only thing is that with the last move, the schedule S' is no longer reduced. But, we know we can transform S' into a reduced form. The reduced form does not increase cache misses, and moreover it agrees with S_{FF} at the first $j+1$ items of the input. This reduced form will be our desired schedule needed to prove the fact.

Exercise 6. The defining property of the Farthest-in-Future Algorithm is that it evicts an item from the cache that is farthest in the future. Where did we use this defining property of the algorithm in our arguments above?