1 Introduction

This lecture covers online algorithms for the online routing problem and the paging problem. For the online routing problem, we show that an exponential penalty based algorithm has a competitive ratio of $O(\log m)$ where $m$ is the number of edges in the graph. For the paging problem, we show that the deterministic Least Recently Used (LRU) strategy is $k$-competitive for a cache of size $k$ and that the randomized version of the 1-bit LRU strategy is $O(\log k)$-competitive.

2 Online routing problem

Recall the online routing problem as introduced in lecture 1. Given a directed or undirected graph $G = (V, E)$ with edge capacities $c(e) \geq 1$, the goal is to route a request at each time step such that the maximum congestion is minimized. More specifically, a routing request from node $s_i$ to $t_i$ with bandwidth $b_i(e)$ is received at every time step $i = 1, \ldots, T$. For each of these requests, an $(s_i, t_i)$-path $P_i$ needs to be selected. Let $\ell_i(e) := \sum_{j=1, e \in P} b_j(e)$ be the load on edge $e$ at time $i$. The congestion on edge $e$ at time $i$ is then:

$$\text{congestion}_i(e) := \frac{\ell_i(e)}{c(e)}$$

The goal is to select paths $P_i$, such that the maximum congestion is minimized at completion:

$$\min \max_{e \in E} \text{congestion}_T(e)$$

Note that the bandwidth $b_i(e)$ is edge dependent. As such, we can assume without loss of generality that all capacities $c(e) = 1$ for $e \in E$ by rescaling the bandwidths and the capacities for each edge. This reduces the problem to $\min \max_{e \in E} \ell_T(e)$.

**Assumption 1** Without loss of generality, the edge capacities $c(e) = 1$ for all $e \in E$.

An intuitive algorithm, say the GREEDY algorithm, is to route the request over the path that creates the smallest increase in congestion. However, we argued in lecture 1 that this algorithm is at least $\Omega(m)$-competitive where $m$ is the number of edges in the graph.

**Algorithm:** Suppose at each time $t$, we maintain a “cost” of the current routing so far as

$$\phi(t) = \sum_{e} \gamma^{\frac{\ell_i(e)}{c(e)}} L^*.$$ 

Here $\gamma > 1$ is to be determined and $L^*$ is optimal offline objective value.
Suppose we route the \(i\)th request along that path from \(s_i\) to \(t_i\) that causes the least rise in cost. That is, minimizes \(\phi(i) - \phi(i - 1)\). Note that this can be done by finding the shortest path from \(s_i\) to \(t_i\), with respect to edge weights:

\[
p_i(e) := \gamma \frac{l_{i-1}(e) + b_i(e)}{L^*} - \gamma \frac{l_{i-1}(e)}{L^*} = \gamma \left( \frac{b_i(e)}{L^*} - 1 \right)
\]

Let us call this algorithm Penalty. Observe that this algorithm has a strong preference for edges with low congestion, but also prefers paths that are shorter due to the additive nature of the penalties.

**Theorem 2 ([1])** The Penalty algorithm with \(\gamma = \frac{3}{2}\) is \(O(\log m)\)-competitive for the online routing problem, where \(m\) is the number of edges in the graph.

Before we prove Theorem 2, note that the penalties \(p_i(e)\) depend on the optimal offline objective value \(L^*\) (which the online algorithm does not know). However, this can be mitigated by estimating \(L^*\) through a geometric series of estimates for \(L^*\). This is also known as a doubling/scaling technique as shown in Theorem 3, and is often very useful (see [4] for other applications).

**Theorem 3** If there exists a \(c\)-competitive algorithm \(A\) for the online routing problem that assumes the offline optimum \(L^*\) is known, then there exists a \(4c\)-competitive algorithm \(A'\) that assumes no knowledge about \(L^*\).

**Proof:** The algorithm \(A'\) proceeds in phases \(\{1, \ldots, p\}\) with a different estimate for \(L^*\) in each phase. Let \(\lambda_k\) be the estimate for \(L^*\) in phase \(k\). Initially, set \(\lambda_0 = \min_{e \in E} b_1(e)\) and at the beginning of phase \(k\), the estimate is doubled, i.e. \(\lambda_k = 2\lambda_{k-1} = 2^k \lambda_0\).

The \(A'\) works as follows. Suppose it is in phase \(k\) and request \(i\) arrives. It computes the path \(P_i\) according to \(A\) with estimate \(\lambda_k\). If there exists an edge \(e \in P_i\) such that \(l_{i-1}(e) + b_i(e) > c\lambda_k\), then the estimate of \(L^*\) is too low. The algorithm enters the next phase \(\lambda_k + 1\) and starts afresh ignoring whatever it did during previous phases, and tries to route request \(i\).

We claim that the congestion of \(A'\) is at most \(4cL^*\). First, note that once \(\lambda_k\) becomes \(L^*\) or more, it is never increased. Thus \(\lambda_p \leq 2L^*\) in the final phase \(p\). As it might route only fewer requests than the offline algorithm \(A\) incurs congest at most \(c\lambda_p \leq 2cL^*\). Moreover by design, during previous phases \(i < p\), the congestion incurred is never more than \(c\lambda_i\), and thus the overall congestion is at most \(\sum_{i \leq p} 2c\lambda_p \leq 4cL^*\).

**Remark 4** To be precise, Theorem 3 requires that the cost of algorithm \(A\) does not exceed \(c\Lambda\) whenever \(\Lambda\) is used as an estimate for \(L^*\).

By replacing \(L^*\) with \(\Lambda\) in the proof of Theorem 2, it is straightforward to show that this holds for the algorithm Penalty.

With Theorem 3 as a tool we are ready to prove Theorem 2.

**Proof of Theorem 2.** Given Theorem 3, it remains to show that Penalty is \(O(\log m)\)-competitive assuming that \(L^*\) is known.
Let $\gamma = 2/3$, and recall the potential $\phi(i) = \sum_{e \in E} \gamma^{\ell_{i}(e)}$. In particular, $\phi(0) = m$ since each edge contributes a value of 1. We will show that $\phi(T) \leq 2m$. If this holds, Theorem 2 directly follows:

$$2m \geq \phi(T) = \sum_{e \in E} \gamma^{\ell_{T}(e)} \geq \max_{e \in E} \gamma^{\ell_{T}(e)}$$

Taking logs on both sides gives $\max_{e \in E} \gamma^{\ell_{T}(e)} \leq L^{*} \log \gamma 2m$.

It remains to show that $\phi(T) \leq 2m$. We show this by bounding the increase in the potential function $\phi(i) - \phi(i - 1)$ at each time step. To simplify notation, we assume without loss of generality that $L^{*} = 1$. The change in the potential function for request $i$ is given by:

$$\phi(i) - \phi(i - 1) = \sum_{e \in P_{i}} \gamma^{\ell_{i-1}(e)} \left( \gamma^{b_{i}(e)} - 1 \right) = \sum_{e \in P_{i}} p_{i}(e)$$

As $\phi(i) - \phi(i - 1)$ is the length of the shortest path with respect to edge weights $p_{i}(e)$, it must be that length of the path $Q_{i}$ as chosen by the optimal offline algorithm (with respect to the online weights $p_{i}(e)$) for request $i$ is at least as large as $\phi(i) - \phi(i - 1)$:

$$\phi(i) - \phi(i - 1) \leq \sum_{e \in Q_{i}} p_{i}(e) = \sum_{e \in Q_{i}} \gamma^{\ell_{i-1}(e)} \left( \gamma^{b_{i}(e)} - 1 \right)$$

Summing over all time steps, we obtain:

$$\phi(T) - \phi(0) \leq \sum_{i=1}^{T} \sum_{e \in Q_{i}} \gamma^{\ell_{i-1}(e)} \left( \gamma^{b_{i}(e)} - 1 \right)$$

Since $L^{*} = 1$, it must be that $b_{i}(e) \leq 1$ for all $e \in E$ and $i = 1, \ldots, T$. Using the fact that $(\gamma^{x} - 1) \leq (\gamma - 1)x$ for all $x \in [0, 1]$, we obtain:

$$\phi(T) - \phi(0) \leq \sum_{i=1}^{T} \sum_{e \in Q_{i}} \gamma^{\ell_{i}(e)} (\gamma - 1) b_{i}(e)$$

$$\leq \sum_{e \in E} \gamma^{\ell_{T}(e)} (\gamma - 1) \sum_{i \in Q_{i}} b_{i}(e)$$

$$\leq \sum_{e \in E} \gamma^{\ell_{T}(e)} (\gamma - 1) \ell_{T}^{*}(e)$$

Here, $\ell_{T}^{*}(e)$ is the offline load on edge $e \in E$ at completion. As $L^{*} = 1$, thus $\ell_{T}^{*}(e) \leq L^{*} = 1$:

$$\phi(T) - \phi(0) \leq \sum_{e \in E} \gamma^{\ell_{T}(e)} (\gamma - 1) = \frac{1}{2} \sum_{e \in E} \gamma^{\ell_{T}(e)} = \frac{1}{2} \phi(T)$$

where we use the value of $\gamma = 2/3$. From the above inequality, we get $\phi(T) - \phi(0) \leq \frac{1}{2} \phi(T)$ and so $\phi(T) \leq 2\phi(0) = 2m$.\endproof
2.1 Intuition behind the Penalty algorithm

It is interesting to note that the Penalty algorithm is also a greedy algorithm with respect to a soft-max function (with the right threshold) instead of the max function.

Given a set of numbers \(a_1, \ldots, a_n\), let the maximum be \(a_{\max} = \max_i (a_1, \ldots, a_n)\). The soft-max with respect to \(\gamma > 1\) of the same set of numbers is given by \(\hat{a} = \log_\gamma \sum_i \gamma^{a_i}\). Observe that \(\hat{a} = \log_\gamma \sum_i \gamma^{a_i} \geq \log_\gamma \gamma^{a_{\max}} = a_{\max}\). Furthermore, observe that \(\hat{a} \leq \log_\gamma (n \gamma^{a_{\max}})\). So \(\hat{a} \in [a_{\max}, a_{\max} + \log_\gamma n]\). In other words, the soft-max provides a relatively tight upper bound on the maximum of a set of numbers.

3 Paging problem

In a system with a fast memory of size \(k\) and a sufficiently large slow memory, it is preferable to request pages from the fast memory. Given a universe of pages \(U = \{1, 2, \ldots, n\}\) and an arbitrary request sequence \(\sigma_1, \ldots, \sigma_m\), the task is to decide which pages to keep in the fast memory. If page \(\sigma_i\) can be served from fast, then no cost is incurred. If page \(\sigma_i\) cannot be serve from fast memory, a page fault occurs. To serve this request, a page needs to be evicted from the fast memory so it can be replaced by \(\sigma_i\). See Table 1 for an example. The goal is to design an eviction strategy that minimizes the number of page faults and thus evictions.

<table>
<thead>
<tr>
<th>Pages in memory</th>
<th>Page requested</th>
<th>Page evicted</th>
<th>Cost for request</th>
</tr>
</thead>
<tbody>
<tr>
<td>1, 2, 3, 4</td>
<td>1</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1, 2, 3, 4</td>
<td>3</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1, 2, 3, 4</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1, 7, 3, 4</td>
<td>4</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1, 7, 3, 4</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2, 7, 3, 4</td>
<td>3</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>2, 7, 3, 4</td>
<td>5</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>2, 7, 3, 5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Example request sequence for the paging problem with cache size \(k = 4\). The set of pages is \(U = \{1, 2, \ldots, 6, 7\}\) and the request sequence is 1, 3, 1, 7, 4, 2, 3, 5.

3.1 Eviction strategies

There are many different eviction strategies. Here are some typical strategies:

- **Least Recently Used (LRU)** - evict the page which last use was furthest in the past.
- **Least Frequently Used (LFU)** - evict the page that has been requested the least.
- **First In - First Out (FIFO)** - evict the page that entered the cache first.
- **Last In - First Out (LIFO)** - evict the page that entered the cache last.
- **Flush When Full (FWF)** - evict any unmarked page and mark the requested page. When all pages are marked, evict all pages from cache.
• Random Replacement (RR) - evict a random page.

Let’s consider the LFU strategy for the paging problem. This strategy is not \(c\)-competitive for any \(c \geq 1\). This can be seen by considering a cache with size \(k = 2\) and the request sequence \((1)^{\ell}(2,3)^{\ell}\) for \(\ell\) sufficiently large. The optimal algorithm will incur cost \(\leq 2\), whereas the number of page faults increases linearly in \(\ell\) for the LFU strategy. A similar argument exists for the LIFO strategy where the request sequence consists of pages \(u, v\) that are repeatedly requested in alternating order [6].

3.2 Best offline strategy

In the offline setting, the request sequence is known in advance. It was shown by Belady that the optimal strategy is to evict the page that will be needed furthest in the future.

Theorem 5 ([2]) Assuming that the full request sequence is known, it is optimal to evict the page whose next request occurs furthest in the future.

Proof: (based on [6]) Let us refer to this strategy as Longest Forward Distance (LFD). Suppose for contradiction that there exists an algorithm SUP superior to LFD. Let \(\sigma = \sigma_1, \ldots, \sigma_n\) be the sequence such that \(SUP(\sigma) < LFD(\sigma)\).

At some point during the request sequence, the algorithms must keep a different cache. Let \(\sigma_d\) be the request that initiates the divergence. Since the cache is different for both algorithms, different pages must have been discarded to service \(\sigma_d\). Let page \(u\) be discarded by SUP and let page \(v\) be discarded by LFD. Let \(t\) be the first time after time \(d\) that algorithm SUP discards page \(v\). Consider a variant of SUP called SUP’ that discards \(v\) instead of \(u\) at time \(d\). Note that SUP and SUP’ converge again after time \(t\).

Since LFD chooses the page that occurs furthest in the future, the next request after time \(d\) for page \(v\) must occur before the next request for page \(u\). Let this request for \(v\) be \(\sigma_t\). Then, the following holds:

Lemma 6 After time \(d\), SUP and SUP’ have \(k - 1\) identical pages in memory. This is maintained until SUP and SUP’ converge at time \(t\). Furthermore, \(SUP'(\sigma) \leq SUP(\sigma)\)

Proof: Case 1 - \(d < t < b\): Consider any page \(i\) for \(d < i < t\). The proof continues by induction on page \(i\). Since \(SUP\) and \(SUP'\) have \(k - 1\) identical pages in memory, there must be a unique page \(x\) in cache for \(SUP'\) but not for SUP.

If \(\sigma_i \neq x\), then SUP and SUP’ are aligned in terms of page faults. Furthermore, \(SUP'\) can evict as done by SUP since page \(v\) is not discarded until time \(t\). If SUP and SUP’ had \(k - 1\) identical pages in cache before request \(\sigma_i\), then the algorithms still share \(k - 1\) identical pages afterwards.

If \(\sigma_i = x\), then only SUP incurs a page fault. A page is evicted by SUP and replaced with \(x\). Once again, \(f SUP\) and \(SUP'\) had \(k - 1\) identical pages in cache before request \(\sigma_i\) then the algorithms still share \(k - 1\) identical pages afterwards.

To fulfill the request \(\sigma_i\), the SUP algorithm evicts \(v\) and the SUP’ algorithm evicts \(x\) such that all \(k\) pages in cache are identical. Furthermore, note that \(SUP'(\sigma) \leq SUP(\sigma)\).

Case 2 - \(b \geq t\): SUP follows the steps described in case 1 for requests \(\sigma_i\) for \(d < i < b\). Following the analysis above, SUP and SUP’ have \(k - 1\) pages in common when \(\sigma_b = v\) occurs. By definition of LFD, it must be that \(u\) occurred before \(v\), so \(\sigma_i = u\) occurred at least
once. Since \( u \) remained in cache for \( SUP' \), only \( SUP \) faulted. Therefore, it must be that \( SUP'(\sigma_1 \ldots \sigma_{b-1}) < SUP(\sigma_1 \ldots \sigma_{b-1}) \). However, \( SUP' \) faults on \( \sigma_b \) and replaces \( e \) with \( v \), whereas \( SUP \) does not. Afterwards, \( SUP \) and \( SUP' \) have identical pages in cache and will follow the same behavior. Furthermore, \( SUP'(\sigma) \leq SUP(\sigma_1 \ldots \sigma) \) since \( SUP' \) had strictly lower cost until time \( b - 1 \) and exactly one more page fault afterwards.

By repeated swapping, \( SUP' \) converges to \( LFD \). Thus, \( LFD(\sigma) \leq SUP(\sigma) \). This proves the statement of Theorem 5.

3.3 Optimal online deterministic algorithms

Based on our knowledge of \( LFD \), we can derive the following hardness result.

**Theorem 7** No online deterministic algorithm for the paging problem can be \( c \)-competitive for \( c < k \) where \( k \) is the cache size.

**Proof:** Let the algorithm \( A \) be an arbitrary online deterministic algorithm for the paging problem. We will construct a request sequence on a universe of \( n = k + 1 \) pages. In particular, the adversary will construct a request sequence by checking which pages algorithm \( A \) will keep in cache and requesting the page that is missing from the cache as the next request. The online algorithm \( A \) will then have a page fault at every request.

In contrast, the offline algorithm will remove the page that is requested furthest in the future. Let the algorithm \( LFD \) have a page fault on request \( \sigma_a \). The evicted page, say \( e \) will have its request furthest in the future. Since we have a cache size of \( k \) pages and only \( k + 1 \) different pages, the only page not in the cache is page \( e \). So the next page fault must be due to page \( e \). Out of the \( k \) pages in the cache just before request \( \sigma_a \), the next request for page \( e \) occurred furthest into the future. So the other \( k - 1 \) pages must occur before page \( e \). As such, there can be at most one page fault per \( k \) requests.

Since the arbitrary online deterministic algorithm \( A \) has a page fault for every request and the \( LFD \) algorithm has a page fault of at most once per \( k \) pages, the competitive ratio of any online deterministic algorithm can be no better than \( k \).

Based on the bound derived in Theorem 7, the best possible online deterministic algorithm would be \( k \)-competitive. Next, we show that the \( LRU \) strategy is \( k \)-competitive. This will be derived by showing that the 1-bit \( LRU \) strategy is \( k \)-competitive and that \( LRU \) is a special case of the 1-bit \( LRU \) strategy.

The 1-bit \( LRU \) strategy is defined as follows:

- Mark a page when it is requested.
- If a page cannot be served from cache, evict some unmarked page and mark the requested page.
- If all pages are marked and a page outside of cache is requested, then unmark all pages, evict some unmarked page, and mark the requested page.
- Initialize with all pages marked.

In particular, we will assume that ties between unmarked pages are broken in the worst possible way. Next, we show that both the 1-bit \( LRU \) and \( LRU \) strategies are \( k \)-competitive.
Lemma 8 The 1-bit LRU strategy is $k$-competitive.

Proof: We divide the requests into phases. Whenever the 1-bit LRU strategy unmarks all pages a new phase is started.

At the end of each phase, all of the pages in the cache have been marked. At the beginning of the phase, none of the pages were marked. Therefore, during the phase there must have been exactly $k$ distinct pages requested. Since a page being marked corresponds to an eviction, the 1-bit LRU algorithm must have made exactly $k$ evictions.

In contrast, the offline optimal LFD algorithm also receives requests for $k$ distinct pages in each phase since the phases are determined purely by the request sequence. Furthermore, each phase must request at least one page $e$ that did not occur in the previous phase. Otherwise, none of the pages would have triggered the start of a new phase for the 1-bit LRU algorithm since all pages occurred and were marked in the previous phase. Hence, the LFD algorithm must evict at least one page to serve the request for page $e$.

In conclusion, the 1-bit LRU performs $k$ evictions per phase and the LFD performs at least one eviction per phase. Hence, the 1-bit LRU strategy is $k$-competitive.

2

Theorem 9 The LRU strategy is $k$-competitive.

Proof: The LRU strategy is a special case of the 1-bit LRU strategy where ties are broken by evicting the unmarked page that was least recently used, so the results from Lemma 8 for 1-bit LRU carry over to the LRU strategy.

Remark 10 It can be shown that the algorithms FIFO and FWF are also $k$-competitive.

3.4 Randomized online algorithms

The best deterministic online algorithms for the paging problem are $k$-competitive. This raises the question of whether a randomized algorithm would work better. In fact, the randomized 1-bit LRU strategy achieves approximately a $O(\log k)$-competitive ratio. The difference with the non-randomized 1-bit LRU strategy is that ties are broken by randomly selecting an unmarked page. Before we formally state and prove this claim, let us first define the competitive ratio in a random context and the various types of adversaries.

3.4.1 Competitive analysis in a random setting

In a random setting, there are different types of adversaries depending on how they observe the outcome of random decisions. These types of adversaries were introduced by Ben-David et al. [3]:

- Oblivious adversary - The oblivious adversary generates the complete request sequence before any requests are served by the online algorithm. The adversary incurs the cost of the optimal offline sequence.

- Adaptive Offline Adversary - Before the adversary creates the request for time $t$, it may first observe the decisions and the outcome of the randomization of the online algorithm for time steps $1, \ldots, t-1$. The adversary incurs the cost of serving the requests online.

7
• **Adaptive Online Adversary** - Before the adversary creates the request for time \( t \), it may first observe the decisions and the outcome of the randomization of the online algorithm for time steps \( 1, \ldots, t - 1 \). The adversary incurs the cost of serving the requests online. It will only receive the decision of the online algorithm after it decided its own response to the request.

The adversaries differ in how the optimal decision is calculated and what information can be incorporated. This directly affects the competitive ratio through the value of \( \text{OPT}(I) \). The competitive ratio for randomized algorithms is defined as follows:

**Definition 11** A randomized algorithm \( \text{ALG} \) is set to be \( c \)-competitive with respect to an adversary of type \( A \) if for all instances \( I \):

\[
\frac{E[\text{ALG}(I)]}{\text{OPT}_A(I)} \leq c
\]

where \( \text{OPT}_A(I) \) is the optimal cost on instance \( I \) incurred by an adversary of type \( A \).

Ben-David et al. also proved the following theorems on how the adversaries relate to each other for any online problem:

**Theorem 12 ([3])** If there exists a randomized online algorithm that is \( c \)-competitive against the adaptive offline adversary, then there exists an online deterministic algorithm that is \( c \)-competitive.

This theorem implies that randomization does not help against the adaptive offline adversary.

**Theorem 13 ([3])** If there exists a randomized online algorithm \( A \) that is \( c \)-competitive against the adaptive online adversary and there exists a \( d \)-competitive algorithm against the oblivious adversary, then \( A \) is \((c \times d)\)-competitive against the adaptive offline adversary.

Together, these theorems imply the following corollary:

**Corollary 14** If there exists a \( c \)-competitive online algorithm against an adaptive online adversary, then there exists a \( c^2 \)-competitive deterministic algorithm.

### 3.4.2 Randomized 1-bit LRU

We will now show that the randomized 1-bit LRU strategy is \( O(\log k) \)-competitive against the oblivious adversary.

**Theorem 15 ([5])** The randomized 1-bit LRU strategy is \( 2H_k \)-competitive against the oblivious adversary, where \( H_k \) is the \( k \)th harmonic number which is approximately equal to \( \log k \).

**Proof:** Similar to the competitiveness proof for the non-randomized 1-bit LRU strategy, we divide the requests into phases. A new phases starts when all requests are unmarked. First, note that the phases are independent from the random decisions since we showed that a phase ends whenever \( k \) distinct pages are requested which depends solely on the request sequence.

Let \( m_i \) be the number of distinct pages in phase \( i \) that were not in phase \( i - 1 \). Let the potential function \( \Phi_i \) be the number of pages that are in cache at the beginning of phase \( i \) for \( \text{OPT} \) but are not in cache for randomized 1-bit LRU. Furthermore, let \( \text{OPT}_i \) be the cost incurred by the randomized 1-bit LRU in phase \( i \). Then,
Lemma 16 $OPT_i \geq m_i - \Phi_i$

**Proof:** There are $m_i$ requests that did not occur in phase $i - 1$ but occur in phase $i$. Since all requests that occurred in phase $i - 1$ are still in memory for the 1-bit LRU algorithm, the offline algorithm can have at most $\Phi_i$ of the $m_i$ requests in memory. Therefore, the offline algorithm occurs a cost of $OPT_i \geq m_i - \Phi_i$ to serve the remaining requests. \hfill $\square$

Lemma 17 $OPT_i \geq \Phi_{i+1}$

**Proof:** The offline algorithm has $\Phi_{i+1}$ pages in memory that are not in the memory of the randomized 1-bit LRU algorithm. Since all pages in the memory of the 1-bit LRU memory have been requested in phase $i$, there must have been $\Phi_{i+1}$ pages that have been evicted out of memory by the offline algorithm and a cost is incurred for each page. \hfill $\square$

Lemma 18 $OPT \geq \frac{1}{2} \sum_i m_i$

**Proof:** Combining Lemmas 16 and 17, we obtain:

$$OPT_i \geq \frac{1}{2} (m_i - \Phi_i + \Phi_{i+1})$$

Let phase $p$ be the last phase. Summing over phases, we obtain:

$$OPT \geq \frac{1}{2} \sum_{i=1}^{p} (m_i - \Phi_i + \Phi_{i+1})$$

$$= \frac{1}{2} \sum_i m_i - \Phi_1 + \Phi_{p+1}$$

Since $\Phi_{p+1} \geq 0$ and by assuming that $\Phi_1 = 0$, the proof of this lemma is completed. \hfill $\square$

Having bounded the cost for the optimal offline algorithm, we now bound the cost of the randomized 1-bit LRU strategy.

Lemma 19 In expectation, the randomized 1-bit LRU algorithm incurs a cost of at most $H_k \sum_i m_i$.

**Proof:** In the worst-case, the request sequence is such that the pages that are not in phase $i - 1$ are requested first, since we may accidentally evict a page that occurs in both phase $i - 1$ and phase $i$. The randomized 1-bit LRU algorithm occurs a cost of $m_i$ for serving the requests that did not occur in phase $i - 1$. Furthermore, an expected cost of $\frac{m_i}{k}$ is incurred for the first request that also occurred in phase $i - 1$, since the $m_i$ out of $k$ pages were replaced by a new request. For the second request that occurred in phase $i - 1$, it must be $\frac{m_i}{k-1}$. And so forth. The total cost incurred is, therefore:

$$\sum_i \left[ m_i + \frac{m_i}{k} + \frac{m_i}{k-1} + \ldots + \frac{m_i}{k-(k-m_i-1)} \right] \leq \sum_i m_i \left[ 1 + \frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{m_i+1} \right] \leq H_k \sum_i m_i$$

Combining Lemmas 18 and 19, we obtain the theorem’s statement. \hfill $\square$
References


