Notes on Online Algorithms (COL 758)

1 List Update Problem

In the list update problem, we need to maintain a set of n elements X in an ordered sequence. The data-structure maintains a pointer to the beginning of the list (call it the head of the list). Starting from the head, it can navigate by moving to the next element and so on. Assume that it can move in 1 time step from the current element to the next or the previous element. Further, the data-structure is also allowed to exchange two consecutive elements in the list – this operation costs 1 time unit. At each time, an online request asks for an element x in the list, and the algorithm incurs the cost of accessing x (i.e., if the element x is p positions from the head of the list, the algorithm incurs cost p for searching for this element). Further, it may incur additional cost for exchanging two consecutive elements.

We analyse the MTF (Move to Front) heuristic: suppose the next request accesses an element x in the list. The algorithm, after searching for x, moves x to the head of the list. So, if the position of x was p (from the beginning), the algorithm incurs p units of time for searching for x, and then performs p exchanges to bring it to the front, i.e., total of $2p$ units. We now show that this algorithm is 4-competitive with respect to any offline adversary. Let $\mathcal{A}$ denote the online algorithm and $\mathcal{O}$ an offline algorithm. Let $L^{\mathcal{A}}(t)$ be the list maintained by the algorithm $\mathcal{A}$ at (the end of) time t, and define $L^{\mathcal{O}}(t)$ similarly. Let $\texttt{cost}^{\mathcal{A}}(t)$ and $\texttt{cost}^{\mathcal{O}}(t)$ denote the cost incurred by $\mathcal{A}$ and $\mathcal{O}$ at time t. In order to analyse the algorithm, we define a potential function $\Phi(t)$. The value of $\Phi(t)$ (at time t) is the number of inversions between the lists $L^{\mathcal{A}}(t)$ and $L^{\mathcal{O}}(t)$. Recall that an inversion is a pair of elements (x, y) such that the ordering of these two elements are opposite in the two lists. Therefore, $\Phi(t)$ could vary between 0 and n^2. Since both $\mathcal{A}$ and $\mathcal{O}$ start with the same initial list, $\Phi(0)$ is 0. Our goal is to show that, for all time $t \geq 1$,

$$\texttt{cost}^{\mathcal{A}}(t) + 2(\Phi(t) - \Phi(t-1)) \leq 4 \cdot \texttt{cost}^{\mathcal{O}}(t). \tag{1}$$

We first see why proving such a statement suffices. Suppose we run the algorithm till time T. Then adding up the above inequality for $t = 1, 2, \ldots, T$, we get $\sum_{t=1}^{T} \texttt{cost}^{\mathcal{A}}(t) + 2(\Phi(T) - \Phi(0)) \leq 4 \cdot \sum_{t=1}^{T} \texttt{cost}^{\mathcal{O}}(t)$. Now, $\Phi(0) = 0$ and $\Phi(T) \geq 0$. So, $\sum_{t=1}^{T} \texttt{cost}^{A}(t) \leq 4 \cdot \sum_{t=1}^{T} \texttt{cost}^{\mathcal{O}}(t)$, which is what we wanted to prove.

We now show that the condition (1) holds at time t. Suppose the element x is requested at time t. We analyse in two steps. We first assume that $\mathcal{O}$ simply accesses x, and then we show that (1). After this we allow $\mathcal{O}$ to exchange two consecutive elements, and we show that this condition continues to hold. Suppose x appears at position p in the list $L^{\mathcal{A}}(t-1)$ and position q in the list $L^{\mathcal{O}}(t-1)$. What is the change in Φ when we move x from position p to the front of the list in $\mathcal{A}$? Let r_1 be the elements which appear before x in $L^{\mathcal{A}}(t-1)$, but after x in $L^{\mathcal{O}}(t-1)$; and r_2 be the elements which appear before x in both $L^{\mathcal{A}}(t-1)$ and $L^{\mathcal{O}}(t-1)$. It is easy to check that $\Phi(t) - \Phi(t-1)$ is $r_2 - r_1$. Further, $r_1 + r_2 = p - 1$ and $q \geq r_2 + 1$. Now, the LHS of the condition (1) is $2p + 2(r_2 - r_1) = 2 + 4r_2 \leq 4q$, which shows that this condition is satisfied. Now, when $\mathcal{O}$ exchanges any two consecutive elements, RHS of this condition changes by 4, but Φ changes by at most 1. Thus, this condition continues to hold.

2 Online Paging and 1-bit LRU

In the online paging algorithm, we have a cache of size k, which can hold up to k pages. When the next page is requested, the algorithm checks if the page is in the cache. If so, it does not incur any cost. But if

the page is not in the cache, it incurs a cache miss. Further, it needs to evict a page from the cache, and bring this requested page in the cache. The objective is to minimise the number of cache misses. We know that if the entire sequence of requested pages is known in advance, "Furthest in Future " is the optimal policy. We first show that no deterministic algorithm can be better than k-competitive.

Consider the following input. There are a set of $k+1$ pages. At each time, we request a page which is not in the cache. Clearly, if we run the algorithm for T time steps, the online algorithm will incur T cache misses. It is also easy to see that "Farthest In Future" will incur about T/k cache misses. Thus, no online algorithm can be better than k-competitive. Now we describe the 1-bit LRU algorithm. Each page in the cache is either marked or unmarked. Initially, all pages are unmarked. When a page x is requested, two cases can happen: (i) the page is already in the cache – in this case, we simply mark this page (if it is already marked, no change happens), or (ii) the page x is not in cache – assuming there is at least one unmarked page in the cache, we evict one of the unmarked pages, and bring in page x in the cache. We also mark this page. If all pages in the cache were marked, we unmark all pages, and then perform the same steps as before (i.e., evict a (unmarked) page, and bring in x, and mark this page).

We show that this algorithm is k-competitive. We divide the timeline into phases. The first phase starts at time 0. Consider a phase i, which starts at time s_i. Let t_i be the first time after s_i when we see $k+1$ distinct pages (during $[s_i, t_i]$). Phase i will end immediately before time t_i, and the next phase will start from time t_i. Thus, phase i can be described by the interval $[s_i, t_i - 1]$.

Exercise 1 *Consider any algorithm (which could be off-line, i.e., even knows the entire future). Show that for each phase i as defined above, either the algorithm incurs a cache miss during phase $[s_i + 1, t_i]$. Conclude that if there are ℓ phases, then any algorithm incurs at least $\ell - 1$ cache misses.*

The above algorithm shows that any off-line algorithm will incur at least $\ell - 1$ cache misses, where ℓ is the number of phases. Now we show that the 1-bit LRU algorithm incurs at most k cache misses during a phase. First of all, it is easy to show by induction (on the number of phases) that if a phase i starts at time s_i, then all pages in the cache are unmarked except for the page which was accessed at time s_i. During a phase, k distinct pages are accessed. When a page p is accessed for the first time in a phase, it may cause a cache miss, but then it gets marked in the cache. Therefore, subsequent accesses to p during this phase will not cause cache misses. Note that when all pages get marked, and a new page is accessed, this phase ends. Thus, the algorithm makes at most k cache misses during any phase. Combined with the above exercise, we see that this algorithm is k-competitive.

We now show that a randomized version of 1-bit LRU has better competitive ratio. The algorithm is essentially same as the 1-bit LRU algorithm above. When the algorithm needs to evict an unmarked page, it picks a randomly chosen unmarked page. We now show that the competitive ratio is $O(\log k)$. Let S_i be the set of k distinct pages accessed during a phase. Note that just before this phase ends, the cache contains S_i (and all these pages are marked at this time). Let n_i denote the pages in S_{i+1} which are not in S_i (call these "new pages") – these are pages which are accessed in S_{i+1}, but not in S_i. First observe that during $S_i \cup S_{i+1}$, $k + n_{i+1}$ distinct pages are accessed, and so, any algorithm will incur at least n_{i+1} cache misses. From this conclude that:

Exercise 2 *Prove that any (even off-line) algorithm will incur at least $\sum_i n_i/2$ cache misses.*

We now show that the randomized 1-bit LRU algorithm incurs expected $O(n_i \log k)$ cache misses during phase i. When a page from $S_i - S_{i-1}$ is accessed for the first time during a phase, it causes a cache miss,

but subsequent accesses to this page will not cause a cache miss (why?). Thus accesses to pages in $S_i - S_{i-1}$ cause n_i cache misses – the exercise above shows that these can be easily accounted for in any algorithm. The worry is about pages accessed during this phase which were also present in S_{i-1} – let these pages be $p_1, \ldots, p_{k-n_i}$, arranged in the order of the first time they get accessed during this phase (again, once a page gets accessed, subsequent accesses to it in a phase do not cause a cache miss). What is the probability that (the first) access to a page p_c causes a cache miss? This page was present (and unmarked) in the cache at the beginning of this phase.

Exercise 3 *Prove that the probability that this access causes a cache miss is at most* $\frac{n_i}{k-c}$.

Using the above exercise, the expected number of cache misses during a phase is $n_i + n_i \cdot \left(1 + \frac{1}{2} + \ldots + \frac{1}{k}\right) = O(n_i \cdot \log k)$.