LRU is optimal on tree access graphs

Theorem 1 *The competitive ratio of LRU on a tree G equals the maximum, over subtrees T of G with at most $k+1$ nodes, of $\ell(T)-1$, where $\ell(T)$ denotes the number of leaves of T.*

The lower bound has been proved for any deterministic algorithm, so that includes LRU. For the upper bound, we track OPT and LRU in parallel during the sequence of requests.

Consider the page r_t requested at time t, plus the set of pages present in LRU's cache right before the request is served. This consists of k or $k+1$ nodes (depending on whether or not there is a page fault) forming a subtree of G. Call this $\text{tree}_t(\text{LRU})$. Note that if LRU has a fault on r_t, then r_t must be a leaf of $\text{tree}_t(\text{LRU})$.

If OPT has a fault on page r_t then we place token on nodes of $T = \text{tree}_t(\text{LRU})$ as follows: for each leaf of T other than r_t (which may or may not be a leaf), follow the path from v to r_t and place a token on the first node that does not yet have a token.

The theorem follows as a corollary of the following two lemmas:

Lemma 1 *When OPT has a fault, at most $\ell(T)-1$ tokens are placed on $T = tree_t(LRU)$*

To prove the first lemma, we first observe that the it is obvious if r_t is a leaf of T, so let's assume that r_t is an internal node of T. Then LRU must not have a fault on r_t, so r_t is in the cache of LRU. Moreover, since r_t is internal it has degree at least 2 in T and so it partitions T into at least 2 subtrees.

Consider the page r_t, plus the set of pages present in OPT's cache right before the request is served. This consists of k or $k+1$ nodes (depending on whether or not there is a page fault) forming a subtree of G. Call this $\text{tree}_t(\text{OPT})$.

Say that a page is *lonely for LRU* if it is in the cache of LRU but not in the cache of OPT. At time t, page r_t is lonely. By the tree structure of G and of $\text{tree}_t(\text{OPT})$, and by the fact that r_t is an internal node of T, it follows that some subtree of T rooted at r_t is entirely absent from the cache of OPT, and so there is a path from some leaf v of T to r_t such that every node on that path is lonely. By the claim below, every lonely page has a token, so when OPT has a fault on r_t no additional token is placed on the path from v to r_t, and so the lemma holds.

Claim 1 *Every lonely page has a token.*

The claim is proved by induction over time. Initially (right after the first k distinct pages have been requested in the sequence) OPT and LRU have the same cache contents, so there is no lonely page and the claim holds.

For the inductive step, consider how a page u becomes lonely for LRU: It must have been in the cache of LRU and of OPT, and have been evicted by OPT to make room for some request r_t. Upon that eviction, if u is a leaf of $\text{tree}_t(\text{LRU})$ then u receives a token and the claim holds, so let's assume that u is an internal node of $T = \text{tree}_t(\text{LRU})$.

By the tree structure of G and of $\text{tree}_t(\text{OPT})$, and by the fact that u is an internal node of T, it follows that some subtree S of T rooted at u is entirely absent from the cache of LRU, so every node on S is lonely. But they were all already lonely right before, so by induction on time every

node in S already has a token, and so, if we take any leaf v of subtree S, the token-placement rule will consider the path from v to r_t and place a token on u. This proves the claim.

Lemma 2 *When LRU has a fault, resulting in the eviction of some page u, u has a token.*

To prove the second lemma, let u be the page evicted by LRU at time t, let $t_1 < t$ be the time when u became the least recently used page, and let $t_0 < t_1$ be the last time u was requested. Consider $T = \text{tree}_{t_1}(\text{LRU})$.Since the next change to T will be the eviction of u, and since evictions always happen to leaves, u is a leaf of T. Assume that u does not have a token. (By the contrapositive of the claim, this implies that u is not lonely, so u is present in the cache of OPT at that time).

What happens between t_1 and t? If OPT has a fault then a token is placed on leaves of T, including u, and we are done. So let's assume that OPT has no faults in $(t_1, t]$. So the request r_t at time t is on a page which is in the cache of OPT but not in the cache of LRU: it is lonely for OPT.

Consider the graph G as rooted at node r_{t_1} and let $T(u)$ denote the subtree rooted at u: since T is a tree containing r_{t_1} and u is a leaf of T, it must be that the nodes of $T(u) - \{u\}$ are all absent from the cache of LRU. Those which are present in the cache of OPT are therefore all lonely for OPT. The following claim gives a converse.

Claim 2 *The pages that are lonely for OPT are all in $T(u)$.*

The claim implies that page r_t, lonely for OPT, is in $T(u)$. From time t_1 to time t, the sequence of requests performs a walk. Because of the tree structure of G, the sequence has to go through a request to page u: contradiction. So it only remains to prove the claim.

The number of pages that, at time t_1, are lonely for OPT equals the number of pages that, at time t_1, are lonely for LRU.

By definition of LRU, if we look at the walk performed by the sequence of requests during the time interval I from time t_0 to time t_1, the walk must visit every node of T and no other node (so that u becomes the least recently used page). By definition of OPT, at time t_0 the cache of OPT contains T, except perhaps for some nodes of T that have never been requested before (and each of those will result in a fault when they are requested during I). Thus: *the number of pages of $T(u)$ that are in the cache of OPT at time t_0 is at least the number of OPT faults during I.*

That number can be decomposed in the faults causing eviction of the page of $T(u)$ from the cache of OPT, and the faults result in some other eviction from the cache of OPT. Now, any page that at time t_1 is lonely for LRU was requested during I, so it was in the cache of OPT at that time, so it must have been evicted by OPT during I. So:

The number of OPT faults during I is at least the number of pages that, at time t_1, are lonely for LRU, plus the number of pages of $T(u)$ evicted by OPT during I.

Combining and rearranging, we deduce that the number of pages that, at time t_1, are lonely for LRU, is at most the number of pages of $T(u)$ that are in the cache of OPT at time t_0 minus the number of pages of $T(u)$ evicted by OPT during I, in other words, the number of pages of $T(u)$ that are in the cache of OPT at time t_1.

This proves the claim.