JOURNAL OF COMPUTER AND SYSTEM SCIENCES 50, 244–258 (1995)

Competitive Paging with Locality of Reference

ALLAN BORODIN*

Department of Computer Science, University of Toronto, and Center for Advanced Studies, IBM Toronto, Canada

SANDY IRANI†

Computer Science Department, University of California, San Diego, California 92037

AND

PRABHAKAR RAGHAVAN AND BARUCH SCHIEBER

IBM T. J. Watson Research Center, Yorktown Heights, NY

Received July 14, 1991; revised April 14, 1992

The Sleator–Tarjan competitive analysis of paging (*Comm. ACM* **28** (1985), 202–208) gives us the ability to make strong theoretical statements about the performance of paging algorithms without making probabilistic assumptions on the input. Nevertheless practitioners voice reservations about the model, citing its inability to discern between LRU and FIFO (algorithms whose performances differ markedly in practice), and the fact that the theoretical comptitiveness of LRU is much larger than observed in practice. In adition, we would like to address the following important question: given some knowledge of a program's reference pattern, can we use it to improve paging performance on that program? We address these concerns by introducing an important practical element that underlies the philosophy behind paging: *locality of reference*. We devise a graph-theoretical model, the *access graph*, for studying locality of reference. We use it to prove results that address the practical concerns mentioned above. In addition, we use our model to address the following questions: How well is LRU likely to perform on a given program? Is there a universal paging algorithm that achieves (nearly) the best possible paging performance on every program? We do so without compromising the benefits of the Sleator–Tarjan model, while bringing it closer to practice.

1. OVERVIEW

This paper deals with the competitive analysis of paging algorithms [11, 22]. A paging algorithm manages a two-level store consisting of a fast memory that can hold k pages, and a slow memory. The algorithm is presented with a sequence of requests to virtual memory pages. If the page requested is in fast memory (a *hit*) it incurs no cost; but if not (a *fault*), the algorithm must bring it in to fast memory at unit cost and decide which of the k pages currently in fast memory to evict in order to make room for it. A paging algorithm is *on-line* it it makes the decision on eviction without knowledge of future requests.

Early work on evaluating paging algorithms focuses on probabilistic analysis [7], assuming that the request sequence is drawn from a probability distribution. Indeed, Franaszek and Wagner [7] compare the page fault rate of LRU and FIFO that of the optimal algorithm for the case where the ihnput sequence is drawn from an arbitrary probability distribution, in the spirit of the competitive analysis of [11, 22]. Under a "worst-case" input any paging algorithm can be made to fault on every request. Sleator and Tarjan [22] proposed an alternative of amortize analysis: let $A(\sigma)$ be the number of faults made by a paging agorithm A on request sequence σ, and let $\mathrm{OPT}(\sigma)$ be the number of faults made by an optimal algorithm (that has the entire sequence σ available to it in advance) [1, 17]. For a finite c, we say that A is a *c-competitive algorithm* [11] if for all σ, $A(\sigma) - c \cdot \mathrm{OPT}(\sigma)$ remains bounded by a constant. For a randomized algorithm A, we replace $A(\sigma)$ by its expectation in the above definition. The *competitiveness* of A, denoted c_A, is the infimum of c such that A is c-competitive, if such a c exists. (If no such c exists then A is said to be non-competitive; all algorithms discussed in this paper are competitive).

The strength of the Sleator–Tarjan style of analysis is that it is more robust than probabilistic analysis, while more practical than worst-case analysis. With these definitions, Sleator and Tarjan showed that no deterministic on-line

* Work supported by NSERC (Canada) and ITRC (Ontario). A portion of this work was done while the author was visiting IBM T.J. Watson Research Center.

† Work supported by an IBM Doctoral Fellowship. A portion of this work was done while the author was visiting IBM T.J. Watson Research Center.

0022-0000/95 $6.00

paging algorithm can achieve a competitiveness les than k and that a number of algorithms used in practice, including *least cently used* (LRU) and *first-in first-out* (FIFO) are k-competitive and thus optimal by this measure.

1.1. *Motivation*

Practitioners voice at least two reservations about these results: (1) the analysis does not make a distinction between LRU and FIFO, whereas in practice LRU is almost always superior to FIFO; (2) the analysis suggests that the number of faults by LRU on a program could be k times the optimal number of faults, whereas in practice [26] the ratio is often much smaller. In general, a competitive upper bound is likely to be received well when the upper bounds proven are small. Further, the approach does not allow us to treat an important practical question: given some knowledge of the memory access patterns of a program, can one use it to tune the paging algorithm for better performance?

We show here that these concerns can be addressed by augmenting the Sleator–Tarjan approach with an important concept used by paging algorithms: the sequence of pages referenced by real programs exhibits a *locality of reference*. It has long been observed [1, 4, 12, 21] that each time a page is referenced by a program, the next page to be referenced is very likely to come from some small set of pages.

In this paper we develop a graph-theoretic model of a program's locality of reference patterns, which we call the *access graph*. We use the model to prove results addressing the above practical concerns, together with a number of other results. We thus provide a theoretical framework for studying the important idea of locality of reference in programs, while retaining the best features of the Sleator–Tarjan measure.

1.2. *The Access Graph Model and Related Previous Work*

The access graph for a program $G=(V, E)$ is a graph in which each node corresponds to one of the pages that a program can reference. The sequence of page references must obey locality constraints imposed by the edges of G: following a request to a page (node) u, the next request must be either to u or to a node v such that (u, v) is in E.

To make the presentation clearer, we concentrate on algorithms that are unaffected by repeated hits on a page (in particular, we do not consider algorithms such as Frequency Count). Thus, we may assume that following a request to u, the next request is to an adjacent node, so that a sequence σ is a walk in the access graph. However, all our results apply just as well to algorithms that do change state on a hit.

Let $A(\sigma)$ be the number of faults made by a paging algorithm A on request sequence σ, and let $\mathrm{OPT}(\sigma)$ be the number of faults made by an optimal algorithm (that has the entire sequence σ available to it in advance) [1, 17]. We say that A is a *c-competitive algorithm on access graph G* if for every walk σ in G, $A(\sigma)-c\cdot\mathrm{OPT}(\sigma)$ remains bounded by a constant. For randomized algorithms, the number of faults $A(\sigma)$ is a random variable. We assume that the sequence σ is specified in advance before the random choices made by A are announced (during excecution); this corresponds to an *oblivious* request sequence [2, 20]. We say that a randomized algorithm A is c-competitive if for all σ, $E[A(\sigma)]-c\cdot\mathrm{OPT}(\sigma)$ remains bounded by a constant. We denote the competitiveness of (on-line) algorithm A with k pages of fast memory on access graph G by $c_{A,k}(G)$. We denote $\min_{A,k}(G)$ (the minimum taken over on-line algorithms A) by $c_k(G)$ and call it the competitiveness for access graph G with k page slots.

For brevity, we refer to the k page slots of fast memory as *servers* [16]. Thus when we speak of moving a server to a page, we mean that the page is brought into that slot in fast memory. However, note that we are not considering the k-server problem [16] here. In particular, the access graph is not to be confused with the graphs determining the distance metrics in the server problem or in metrical task systems [3]; the "distance" traveled by a server between any two nodes of G is unity. Rather, the access graph restricts the request sequences. When G is the complete graph, our model specializes to the Sleator–Tarjan model. Our premise is that access graphs for real programs are not complete graphs and that some of the discrepancies between the Sleator–Tarjan theory and empirical observations can be explained by this. Also for convenience, we refer to the optimal off-line algorithm and the source of requests together as the *adversary*, who generates the sequence and serves the requests off-line. We now give some examples of access graphs and then describe related prior work.

EXAMPLE 1. The access pattern of a program is often governed by the data structures it uses; for instance, the access graph for a program performing operations on a tree data structure is likely to resemble a tree, whereas for a program doing picture processing or matrix computations it is likely to resemble a mesh. Tree access patterns arise in many important applications: in data structures for key storage and retrieval, in game-tree evaluation, and in branch-and-bound algorithms. In Section 3 we show that LRU is optimal on every tree and that FIFO performs badly, even on trees. We also show that LRU is at most a factor of $\frac{3}{2}$ from optimal in meshes.

EXAMPLE 2. Let G be a cycle on $n=k+1$ nodes. It is easy to see that $c_{\mathrm{LRU},k}(G)$ and $c_{\mathrm{FIFO},k}(G)$ are both k in this case—the sequence that goes clockwise continuously causes both algorithms to fault on every request, whereas the optimal algorithm need fault only once in k requests. On the

other hand, $c_k(G) = \lceil \log(k+1) \rceil$.[1] Consider an algorithm that operates in *phases*: whenever a node is requested during a phase, the node is marked. Thus the set of unmarked nodes at any time in a phase forms a path; on a fault, the algorithm serves with the server at the mid-point of this path. If on a fault there is no server on an unmarked node, the phase is declared over and all nodes are unmarked. Clearly the adversary incurs at least one fault in each phase, and our on-line algorithm at most $\lceil \log(k+1) \rceil$. The lower bound uses the same ideas. Thus both LRU and FIFO perform poorly on this access graph (a fact observe in practice—LRU and FIFO perform poorly on loops just larger than k, leading the designers of the ATLAS computer [12] to develop a paging algorithm with a "loop detector"). Here both $c_{\mathrm{LRU},k}(G)$ and $c_{\mathrm{FIFO},k}(G)$ are far larger than $c_k(G)$. In Section 4 we give on-line paging algorithms that are close to optimal on every undirected access graph (including ones with loops).

EXAMPLE 3. The control flow of a program affects its access graph: the access graph for a program might look like a series-parallel graph with loops (possibly nested) hanging off. The path taken by an execution and the number of times around each loop are data-dependent; we model these by the adversary's choices. In such a case, when the control flow of the program determines the access graph ("structural locality" [9]), it is important to consider digraphs. This is the subject of Section 5.1.

The literature of computer performance modeling and analysis contains much related work, both theoretical and empirical. Shedler and Tung [21] were the first to propose a Markovian model of locality of reference. Spirn [23] gives a comprehensive survey of models for program behavior, focusing on semi-Markovian methods for generating sequences exhibiting locality of reference. Denning [4] (and references therein) develops the *working set* model of program behavior for capturing locality of reference. Other researchers have extended this approach, using it for at least two purposes: (1) to compare the performances of different paging strategies and to tune paging parameters [9, 15]; and (2) to improve program behavior by restructuring programs [5, 8], so that program blocks and data are packed into virtual memory pages so as to ensure good locality of reference. While our focus here will be on the first of these goals, our model and some of our results are likely to prove useful in studies of the latter problem.

Typical questions we study using our model are: (1) How do LRU and FIFO compare on different access graphs? (2) Given the access graph model of a program, what is the performance of LRU on that program (i.e., what is $c_{\mathrm{LRU},k}(G)$)? (3) Given an access graph G, what is $c_k(G)$? Can a good paging algorithm be tailor-made given an access graph G? (4) Is there a "universal" algorithm whose competitiveness is close to $c_k(G)$ on every access graph? By Example 2, LRU and FIFO are not condidates. (5) What is the power of randomization in the access graph model?

[1] The base of all logarithms in the paper is two.

1.3. *Outline of Results*

In Section 2 we give several general results for paging with access graphs. We first show that, given G, determining the competitiveness $c_k(G)$ for G is computable. We then give lower bounds on $c_k(G)$, first using a spanning tree characterization and then using a more sophisticated graph decomposition we call a *vine decomposition*.

Section 3 is an in-depth study of the LRU (least-recently used) algorithm. The main results (Theorems 9 and 10) determine $c_{\mathrm{LRU},k}(G)$ for every G and k to within a factor of two (plus additive constant). Our technique uses combinatorial properties of small subgraphs of G involving the number of articulation nodes in each subgraph. Another useful way of viewing our tight bounds for LRU is as a characterization of "bad" access graphs for LRU. A refinement of our analysis shows that LRU is optimal among on-line algorithms for the important special case when G is a tree (Example 1). We also give results showing that for small k, LRU is close to optimal on every G; these results are of great interest in *caching*, where k is often small (e.g., two or four).

Section 4 addresses the question of a universal algorithm whose competitiveness is close to $c_k(G)$ for every k and G. We show that an extremely simple and natural algorithm (FAR) achieves a competitiveness within $O(\log k)$ of $c_k(G)$ for all k and G, a performance that LRU or FIFO cannot match. We then show that a different algorithm (TS) comes within a constant factor of $c_k(G)$ when $n = k+1$. This special case ($n = k+1$) is important because it is a well-studied case [16, 20, 22] that often provides important insights into the performance of an algorithm when $n > k+1$; indeed all known lower bounds for on-line paging and server systems are proved with $n = k+1$. Recently, Irani *et al.* [10] have shown that our algorithm FAR comes within a constant factor of $c_k(G)$ for all k and G.

Section 5.1 deals with an important class of directed access graphs which we call *structured program graphs*: access patterns likely to arise in the execution of programs written in a structured language with loops and branches, but no GOTOs. This access model is especially interesting for studying instruction caches. We analyze a simple and natural algorithm; one conclusion we draw is that our algorithm is optimal when $n = k+1$.

Section 5.2 presents some results on randomized algorithms: we show that a variant of an algorithm of Fiat *et al.* [6] is within a constant factor of optimal whenever G is a tree, and also we show that there are access graphs for

which randomization cannot improve the competitiveness by more than a constant factor. We conclude with a number of open problems and directions for further work.

2. GENERAL RESULTS FOR PAGING WITH ACCESS GRAPHS

2.1. *The Complexity of Determining the Competitiveness*

Let $G=(V, E)$ be an access graph (directed or undirected). Any on-line paging algorithm A with a set S of states managing k pages is mapping $V \times V^k \times S \to V^k \times S$ satisfying the condition that $A(u, \langle v_1, ..., v_k \rangle, s) = (\langle v_1', ..., v_k' \rangle, s')$ implies $u \in \{v_1', ..., v_k'\}$. That is, given a page request u, the next *configuration* of fast memory pages (= servers) must include u. We can extend the mapping to request sequences in the obvious way. A request sequence $\sigma = u_1, u_2, ...$ is admissible if for all i, $u_i = u_{i+1}$ or $\langle u_i, u_{i+1} \rangle \in E$. Below, we consider extensions of the mapping only to admissible sequences. Before developing the tools for analyzing specific paging algorithms, it would be helfpful to know that the synthesis of optimal on-line paging algorithms is, at least theoretically, achievable. The following results are similar in spirit to decidability results of McGeoch *et al.* [19] for server problems.

PROPOSITION 1. *In general, it is undecidable if a given paging algorithm A achieves a given competitive ratio.*

Proof. For every i, we could construct an algorithm A_i which follows a known competitive algorithm on the jth request if the ith Turing machine on input i halts in at most j steps, else it follows a known algorithm with no finite competitive ratio. ∎

PROPOSITION 2. *For any k, any finite access graph $G=(V, E)$ and any finite state algorithm A (i.e., $|S|$ is bounded by a computable function of $n=|V|$), we can compute $c_{A,k}(G)$ in space polynomial in n and* $\log |S|$. *(All the paging algorithms we consider are indeed finite state.)*

Proof. Because $|S|$ is finite, there must be a reachable configuration/state pair $\langle v_1, ..., v_k \rangle$ and s, and a finite length sequence σ with $A(\sigma, \langle v_1, ..., v_k \rangle, s) = (\langle v_1, ..., v_k \rangle, s)$ such that the adversary can extract the ratio $c_{A,k}(G)$ by forcing A into $(\langle v_1, ..., v_k \rangle, s)$ and then repeating the sequence σ. ∎

PROPOSITION 3. *For any k, and any finite access graph G, the competitive ratio $c_k(G)$ is computable. Furthermore, a finite-state on-line algorithm B which realizes this competitive ratio is constructible.*

Proof. Since $c_k(G)$ is monotone in the addition of edges to G, and since $c_k(K_n)$ is k for all n, we have that k is an upper bound on $c_k(G)$ for any G. Using this fact and the observations of Manasse *et al.* [16] concerning p-residues, we can restrict attention to finite-state algorithms whose state set S is a subset of $[-(k+1)\cdot k, (2k+1)\cdot k]^{n^k}$. Again, following [16], we can consider all possible ways to serve a request when in a given configuration and state. Noting that every admissible request sequence must result in a repeated configuration/state pair within $n^k(3k^2+2k)^{n^k}$ requests, we construct the algorithm B which minimizes the ratio between B and the off-line dynamic programming algorithm. The minimum is taken over all admissible request sequences that cause B to cycle when started in a reachable configuration/state. ∎

2.2. *Lower Bounds on the Competitiveness*

Given an access graph G, we have observed that $c_{A,k}(G)$ and $c_k(G)$ are monotone in the addition of edges to G; thus $c_{A,k}(G) \leq c_{A,k}(K_n)$, where $n=|V|$. Also, a lower bound on $c_{A,k}(G)$ or $c_k(G)$ can be obtained by deleting edges in G. Let $\mathcal{T}_i(G)$ denote the set of trees on i nodes in G. For a tree T, let $l(T)$ denote the set of leaves of T. From the above observation and from ideas similar to those in [22] we have

PROPOSITION 4. *For any G, k and any on-line algorithm A, $c_{A,k}(G) \geq \max_{T \in \mathcal{T}_{k+1}(G)} |l(T)| - 1$.*

The lower bound of Proposition 4 is forced by an adversary who restricts the set of requests to walks on a tree T in $\mathcal{T}_{k+1}(G)$. The walk can always proceed from its present position to any leaf in $l(T)$ without passing through any other leaf, so that an argument similar to that in [22] can now be invoked on these leaves. Note that this lower bound is weak on some graphs, e.g., when G is a cycle on $k+1$ nodes. (Here Proposition 4 only provides a constant lower bound, whereas by Example 2 we know that $c_k(G)$ is $\Theta(\log(k))$. We now remedy this with a more sophisticated graph-theoretic construction.

A *vine decomposition* $\mathcal{V}(H)=(T, \mathcal{P})$ of any graph H is a tree T in H, together with a set of paths $\mathcal{P}=P_1, P_2, ...$, such that (i) the end-points of each of the paths are nodes in T, the internal nodes are not in T; (ii) the internal nodes of the paths are disjoint; (iii) the union of the nodes in T and on the paths gives the node set of H.

Let P_1 be the longest path and let n_T be the number of leaves of T not on any path. The *value* of a path P, denoted $v(P)$, is defined to be $1+\lfloor \log |P| \rfloor$. (The length of a path P is the number of its edges.) Let $\mathcal{H}_x(G)$ denote the set of connected node-induced subgraphs of G containing x nodes.

THEOREM 5.

$$c_k(G) \geq \max_{1 \leq g \leq k} \max_{H \in \mathcal{H}_{k+g}(G)} \left\{ \max_{T \in \mathcal{V}(H)} \left(\frac{v(P_1) - \lceil \log g \rceil}{2} + \frac{\sum_{i>1} v(P_i) + n_T}{g} \right) \right\} - 1.$$

Proof. Without loss of generality we assume that G is not a tree (or else $\mathscr{P}$ is empty and the bound degenerates to that of Proposition 4). Suppose that we are given an on-line algorithm A. We show an adversarial sequence of requests that forces A to achieve a competitiveness greater than or equal to the desired lower bound. The adversary concentrates on a subgraphs $H \subseteq G$ achieving the maximum of the lower bound expression. We claim that in H, $|P_1| > g$. Otherwise, the subgraph $H' \subseteq H$ with $k + g'$ nodes, given by repeatedly removing nodes on paths other than P_1 until $|P_1| > g'$ would give a better bound.

Since H contains $k + g$ nodes, there are g "holes" (nodes where A does not have a server) in H at any time. We partition the request sequence into phases. A phase starts with A's and the adversary's servers aligned and all of the nodes unmarked. A node becomes marked when it is requested.

The rules according to which the adversary generates the next request are as follows. If A has a hole on a marked node, the adversary requests this hole. In case the previous request is not adjacent to this hole, the adversary requests all the nodes in a path from the previous request to the hole that consists only of marked nodes. (Such a path always exists since the marked nodes form a connected component.) Suppose that there is no hole on a marked node. If A has a hole on a leaf x of the tree, the adversary requests all the nodes in a path from the previous request to x that consists only of marked nodes and interior tree nodes. If A has a hole on a path P_i, for $i > 1$, the adversary requests all the nodes in a path to the hole that consists only of marked nodes, interior tree nodes, and at most one half of the unmarked nodes in P_i.

Suppose that A has all of its holes on unmarked nodes in P_1. As long as the number of unmarked nodes in P_1 is at least $2g$, the adversary can hit at least $g/2$ holes by requesting half of the unmarked potion of P_1 that has the most holes. In the process the adversary only requests marked nodes, interior tree nodes, and half of the unmarked nodes in P_1. If the number of unmarked nodes in P_1 is less than $2g$ (but at least g), the adversary can hit one more hole requesting all but g unmarked nodes.

The adversary continues to hit A's holes in this manner until the end of the phase, when all but g nodes have been marked. The adversary services the sequence by initially moving all its g holes to the g nodes that are left unmarked at the end of the phase. Thus, the adversary incurs no more than g faults per phase. The adversary requests all marked nodes until the servers are aligned. The phase ends, and all the nodes are declared unmarked.

Except for the g nodes left unmarked at the end of the phase, A incurs a fault for every leaf node that is not on any path, because such a leaf node is marked only when it is requested while A has a hole there. Each path P_i, for $i > 1$, is hit $v(P_i)$ times, and A incurs a fault each time. In at least $v(P_1) - \lceil \log g \rceil$ of the hits on P_1, A incurs at least $g/2$ faults. Thus, A incurs at least $(v(P_1) - \lceil \log g \rceil)\, g/2 + n_T + \sum_{i>1} v(P_i) - g$ faults in a phase. ∎

Irani *et al.* [10] have shown that for any graph G there always exists a subgraph H which either has $k + 1$ nodes or is a single cycle that achieves the maximum of the expression in Theorem 5 to within a constant factor.

2.3. *Useful Properties of Paging Algorithms*

Many of the algorithms studied in this paper are *marking* algorithms [6, 11]. A marking algorithm proceeds in phases. At the beginning of a phase all the nodes are unmarked. Whenever a node is requested, it is marked. On a fault, the marking algorithm vacates a server from an unmarked node (chosen by a rule specified by the algorithm) and brings it to the request. A phase ends at the first fault after every server is on a marked node (equivalently, a phase ends when k different nodes have been requested during the phase). At this point all the nodes become unmarked and a new phase begins. We use the following proposition to analyze the competitive ratio of marking algorithms.

PROPOSITION 6. (a) *If g_i is the number of nodes requested in the ith phase that were not requested in the $(i-1)$th phase, then the cost of the adversary during the first i phases is at least $(\sum_{j=1}^{i} g_j)/2$.* (b) *If A is a marking algorithm then for any graph G, $c_{A,k}(G) \leqslant k$.*

The proof of part (a) is due to Fiat *et al.* [6]. The proof of part (b) is due to Karlin *et al.* [11].

The following result of Belady [1] and Mattison *et al.* [17] (see also [18, 23]) will also prove useful. Consider the off-line paging algorithm that, given an entire request sequence σ, uses the following policy: on a fault, it evicts the item in fast memory whose next access occurs furthest in the future in σ. This algorithm is called OPT, and the reason is evident from the following proposition.

PROPOSITION 7. *For every request sequence σ, the cost of algorithm OPT is the same as the optimal cost.*

3. ANALYSIS OF LRU

3.1. *LRU on General Access Graphs*

We analyze LRU's competitiveness by defining two parameters of the graph G. One is used to lower bound $c_{k,\mathrm{LRU}}(G)$ and the other is used to upper bound $c_{k,\mathrm{LRU}}(G)$. Moreover, we prove that for any graph G, the two parameters differ by at most a constant factor.

LRU has the property that it maintains all of its servers in one connected subgraph of the access graph at all times. Given a node-induced subgraph $H \subseteq G$ with $k + g$ nodes, let v be an articulation node in H; that is, the removal of v from H separates H. Consider the components obtained by

removing v. If no such component contains at least k nodes and if LRU's servers are completely contained in H, then v cannot be a hole, or otherwise LRU's servers would no longer be connected.

Let $\alpha(H)$ denote the number of articulation nodes in H. Let $\beta(H)$ denote the number of articulation nodes in H, whose removal separate H into components none of which contains at least k nodes.

Define

$$a(G) = \max_{k \geq g \geq 1} \{ \max_{H \in \mathcal{H}_{k+g}(G)} (k+g-\alpha(H)-1)/g \}$$

$$b(G) = \max_{k \geq g \geq 1} \{ \max_{H \in \mathcal{H}_{k+g}(G)} \min\{k, k+g-\beta(H)\}/g \}.$$

We prove the following three theorems.

THEOREM 8. $b(G) \leq a(G) + 1$.

THEOREM 9. $c_{\mathrm{LRU},k}(G) \leq 2b(G)$.

THEOREM 10. $c_{\mathrm{LRU},k}(G) \geq a(G)$.

Proof of Theorem 8. Let H be a subgraph of G with $k+g$ nodes. We show that there exists a subgraph $H' \subseteq H$ with $k+g'$ nodes, for which $1 + (k+g'-\alpha(H')-1)/g' \geq \min\{k, k+g-\beta(H)\}/g$. The theorem follows.

There are two cases:

Case 1. The graph H has a biconnected component H' with $k+g' > k$ nodes. In this case $\alpha(H') = 0$, and $(k+g'-1)/g' \geq k/g' \geq k/g \geq \min\{k, k+g-\beta(H)\}/g$.

Case 2. The graph H has no biconnected components with more than k nodes. If $\alpha(H) = \beta(H)$, then $1 + (k+g-\alpha(H)-1)/g \geq (k+g-\beta(H))/g \geq \min\{k, k+g-\beta(H)\}/g$. Suppose that $\alpha(H) > \beta(H)$. Let $\mathcal{B}(H)$ be the set of articulation nodes in H whose removal separates H into components none of which contains k nodes (i.e., $|\mathcal{B}(H)| = \beta(H)$). Let C be the subgraph induced by the set of nodes in the biconnected components of H that contain a node from $\mathcal{B}(H)$. Note that C is a connected component of H and that its articulation nodes are the nodes in $\mathcal{B}(H)$.

The subgraph H' is defined recursively. Initially, H' is set to H. As long H' contains an articulation node v that disconnects H' into components (each including a copy of v) one of which is of size at least k, set H' to be this component. (Note that this component contains C).

Suppose that at the end of the process H' has $k+g'$ nodes. Our goal is to give a lower bound on the number of nodes that are not articulation nodes in H'. Let Γ be the set of articulation nodes in H' that are contained in C, but are also contained in a biconnected component not in C. These are all the nodes in H' that are adjacent to a node in C and a node not in C.

Consider a node $v \in \Gamma$. Since $v \notin \mathcal{B}(H)$, the removal of v from H separates H into components (each including a copy of v), one of which is of size at least k. (Note that this component contains C.) Call this component the "big" component of v in H and call the rest of the components the "small" components of v in H. The total size of the "small" components is at most $g-1$. Define the "big" component of v in H' to be the subgraph of the "big" component of v in H induced by its nodes that are also in H'. Define the "small" components of v in H' similarly.

Let u and v be any two nodes in Γ. Note that the "small" components of v in H' are contained in the "big" component of u in H'. This implies that the "small" components of u and v are disjoint.

Let v be a node in Γ. Each "small" component of v in H must contain at least one biconnected component with only one articulation node. The size of this component is at least $g'+1$; otherwise H' contains an articulation node that disconnects H' into components one of which is of size at least k. Thus this biconnected component contains at least g' non-articulation nodes.

Now, we give a lower bound on the number of non-articulation nodes in H'. The number of non-articulation nodes in C is $|C| - \beta(H) - |\Gamma|$. The total number of nodes in the "small" components of nodes in Γ in H' is $k+g'-|C|+|\Gamma|$. For $v \in \Gamma$, the size of the "small" components of v in H' is at most g. Thus, $|\Gamma| \geq (k+g'-|C|+|\Gamma|)/g$. For each node in Γ we have at least g' nodes that are non-articulation nodes. We conclude that the number of non-articulation nodes in H' is at least $|C| - \beta(H) - |\Gamma| + (k+g'-|C|+|\Gamma|)\, g'/g$.

We obtain

$$\begin{aligned} 1 + \frac{k+g'-\alpha(H')-1}{g'} &\geq 1 + \frac{|C|-|\Gamma|-\beta(H)-1}{g'} + \frac{k+g'-|C|+|\Gamma|}{g} \\ &\geq 1 + \frac{|C|-|\Gamma|-\beta(H)-1+k+g'-|C|+|\Gamma|}{g} \\ &\geq \frac{k+g-\beta(H)}{g}. \end{aligned}$$

∎

Proof of Theorem 9. Break the sequence into phases. A phase ends when requests to k different nodes have been seen. Let g be the number of nodes requested in the current phase that were not requested in the previous phase. By Proposition 6, the cost of the optimal algorithm for the current phase is at least $g/2$, amortized. Consider the set of $k+g$ nodes that are requested in either the current or the previous phase. Let H be the subgraph induced by these $k+g$ nodes. The number of faults that LRU incurs in the phase is bounded by k. Furthermore, the number of faults that LRU incurs in the phase is also bounded by $k+g-$

$\beta(H)$, because LRU does not vacate an articulation node whose removal results in components of size less than k. ∎

Proof of Theorem 10. Let H be any subgraph of G on $k+g$ nodes. We prove that $c_{\mathrm{LRU},k}(G) \geq (k+g-\alpha(H)-1)/g$. Without loss of generality, we assume that no articulation node in H separates it into components one of which consists of less than g nodes. Otherwise, let H' be a subgraph of H with $k+g'$ nodes given by removing one such component. Clearly, $(k+g'-\alpha(H')-1)/g' \geq (k+g-\alpha(H)-1)/g$, and thus proving for H' suffices.

In the proof we use two graph theoretic notions: *ear decomposition* and *st-numbering*. First, we review their definitions.

An ear decomposition [24] of a graph G, starting at a specified edge e is a decomposition of the edges of G into simple paths (ears) $P_0, P_1, ..., P_m$, where P_0 is e, and for each path P_i, $i>0$, each of its endpoints is contained in some P_j, $j<i$, but none of its internal nodes is contained in such an ear. An ear decomposition is *open* if none of the ears is a cycle. An ear P_i, $i>0$, is *trivial* if it consists of a single edge. A graph is biconnected if and only if it has an open ear decomposition starting from any edge. A graph is biconnected if and only if it has an open ear decomposition starting from any edge.

An *st*-numbering of a graph G with n nodes and a specified edge (s, t) is a numbering of the nodes of G with distinct numbers in the range $[1, ..., n]$ with the following properties: (i) node s is numbered one and node t is numbered n; and (ii) every node than s and t are adjacent to a lower numbered node and a higher numbered node. In the sequel we refer to nodes by their *st*-number. The following propositions follow from the properties of *st*-numbering.

PROPOSITION 11. *For each $i>1$, there is a path between node $i-1$ and node i that only contains nodes numbered less than i.*

PROPOSITION 12. *For each $i<n$, there is a path between node $i+1$ and node i that only contains nodes numbered greater than i.*

We return to the proof. Given H, we compute a open ear decomposition of one of its biconnected components that contains only one articulation node. (Such a biconnected component must exist.) Consider the non-trivial ear with the highest index. Without loss of generality we may assume that this ear contains at least g intenal nodes. Otherwise, we may concentrate on the subgraph $H' \subset H$ given by removing these internal nodes and obtain a better bound. Call this ear the "start" path of H. Note that none of the nodes on this path are articulation nodes.

Define a new graph F as follows. The set of nodes of F consists of one node for each non-articulation node of H and two nodes for each articulation node of H. Two nodes u and v in F are connected by an edge if the nodes in H corresponding to u and v are connected by an edge. Now, remove from F all edges that correspond to trivial ears with endpoints in the "start" path. Note that F is biconnected and contains $k+g+\alpha(H)$ nodes. Moreover, since we removed all the trivial ears, the degree of all nodes in F that correspond to nodes in the "start" path (except the last) is two. Later, when no confusion may arise, weidentity the nodes in H and in F.

Since F is biconnected, it has an *st*-numbering [14], where s is (the node corresponding to) the first internal node in the "start" path and t is its adjacent endpoint.

The adversary works in phases. At the beginning of a phase both LRU and the adversary have their holes on nodes $\{1, ..., g\}$, and their servers on the rest of the nodes of H. (Nore that the nodes numbered one to g in F each corresponds to a distinct node of H.) A phase consists of two sub-phases. In each sub-phase the adversary incurs g faults while LRU incurs at least $k+g-\alpha(H)-1$ faults.

In the first sub-phase, the adversary starts by requesting nodes that have servers on them in order to fix the order in which the servers were last used. Specifically, the adversary requests nodes $g+1, g+2, ..., k+g-\alpha(H)$. Going from node i to node $i+1$ the adversary requests only nodes whose number is greater than i. As a result of this, node $g+i$ is the ith least recently used node. Now, the adversary requests node one which is connected to node $k+g+\alpha(H)$, where the previous request was, and then, nodes $2, ..., k+g'-1$, where $k+g'$ is the non-articulation node with the highest *st*-number. This time, going from node i to node $i+1$ the adversary requests only nodes whose number is less than i.

We analyze the number of faults incurred by the adversary and LRU in this subphase. The adversary incurs g famults, by serving nodes one to g using the server currently on node $k+g'$. Note that since the degree of nodes one to $g-1$ is two, the adversary can avoid nodes one to $g-1$ while hitting nodes $g+1, ..., k+g'-1$.

Because of the ordering of the nodes, LRU would incur one fault for each node in H that corresponds to a node numbered in the range $[1, k+g'-1]$ in F that is not an articulation node, totalling $k+g-\alpha(H)-1$.

Before starting the second sub-phase the adversary requests nodes $g, ..., k+g'-1$ repeatedly, so that both LRU and the adversary have their holes on nodes $\{k+g'\} \cup \{1, ..., g-1\}$. The second sub-phase is symmetric to the first one. The adversary requests nodes $k+g'-1$ down to g. Going from node $i+1$ to node i the adversary requests only nodes whose number is less than i (but at least g). As a result of this, node $k+g'-i$ is the ith least recently used node. Now, the adversary requests nodes $g-1, ..., 1, k+g+\alpha(H), ..., g+1$. This time, going fom node $i+1$ to node i the adversary requests only nodes whose number is greater than i.

Again, the adversary incurs g faults, by serving nodes $g-1$ down to 1 and $k+g'$ using the server currently on g. Because of the ordering of the nodes, LRU would incure one fault for each node in the ranges $[1, g-1]$, and $[g+1, k+g+\alpha(H)]$ that is not an articulation node, totalling $k+g-\alpha(H)-1$. ∎

3.2. *LRU on Trees and Meshes*

Let G be a tree. Then, $a(G)$, the lower bound for $c_{\mathrm{LRU},k}(G)$, becomes $\max_{T\in\mathcal{T}_{k+1}(G)}|I(T)|-1$. By Proposition 4 this is also the lower bound for any on-line algorithm. For this important class of access graphs, we show that LRU achieves this lower bound, implying that LRU is optimal.

THEOREM 13. *When G is a tree, $c_{\mathrm{LRU},k}(G)=a(G)$.*

Proof. Let *LRUtree*(t) be the tree formed by the nodes occupied by LRU's servers and the next node requested at time t. To prove the theorem we use a charging scheme. In this scheme, at each point of time, some of the LRU servers will have *tokens*. Whenever, an LRU server services a fault, it has to "pay" a token.

Suppose that OPT faults at time t. Consider the tree formed by *LRUtree*(t). For every leaf in this tree that is not the requested node, consider the path from the leaf to the requested node. Place a token on the first node on the path without a token (if there is such a node). If an LRU server services a fault, then throw away its *token*. Theorem 13 follows from the following two lemmata.

LEMMA 14. *At most $a(G)$ tokens are placed for any OPT fault.*

LEMMA 15. *No LRU server without a token will service an LRU fault.* ∎

Before proving the lemmata, we need a few facts.

PROPOSITION 16. *If G is a tree, then OPT maintains its servers in a connected subgraph of G.*

Proof. By induction on the number of requests. Suppose that up to request t, OPT's servers are in a connected subgraph of G. Let *OPTtree*(t) denote the tree formed by the nodes occupied by OPT servers just before time t, together with the node of the tth request. If OPT faults at time t, then it evicts the server that is on the node whose next request occurs fathest in the future. This is always a leaf of *OPTtree*(t). Thus OPT's servers will be connected after time t. ∎

An LRU server is *lonely* if the node it occupies is not occupied by an OPT server. Similarly, an OPT server is *lonely* if the node it occupies is not occupied by an LRU server.

PROPOSITION 17. *Every lonely LRU server has a token on it.*

Proof. Consider a node v with both an LRU and an OPT server such that after a request at time t, the LRU server will be lonely. If the LRU server on v already has a token, we are done. Otherwise, let r be the requested node. Assume that up until this point in time, all lonely LRU servers have tokens. OPT is about to incur a fault because the OPT server is vacating v. Since v is a leaft of *OPTtree*(t) and a node in *LRUtree*(t), there is some leaf l of *LRUtree*(t) such that the path from l to r passes through v. Furthermore, v is the first non-lonely node on this path, and hence v is the first node along the path from l to r without a token. Thus v gets a token just before the request. When an LRU server services a request, it loses its token, but then it is no longer lonely. ∎

We now turn to prove the lemmata.

Proof of Lemma 14. Suppose OPT faults at time t. If the requested node has no LRU server, then it must be a leaf in *LRUtree*(t) and it is clear that the number of tokens placed after the next fault is no more than $a(G)$. Suppose that the requested node has an LRU server. To prove that no more than $a(G)$ tokens are placed, we have to show that there is one leaf in the tree for which no token is placed also when the requested node is an interior node of *LRUtree*(t). Since the requested node is a leaf of *OPTtree*(t), there is a path containing only nodes with lonely LRU servers from the requested node toa leaf of *LRUtree*(t). No token will be placed for this leaf because all the servers on this path have tokens. ∎

Proof of Lemma 15. Suppose some LRU server with no token becomes the least recently used server at time t_1. The node where the server is located, v, is a leaf of *LRUtree*(t_1). Let $T(v)$ be the subtree of G rooted at v that does not contain any other LRU server. For the proof we need the following proposition.

PROPOSITION 18. *At time t_1, all of the lonely OPT servers are in $T(v)$.*

Proposition 18 implies Lemma 15 because at the next LRU fault, either OPT will incur a fault (and the server on v receives a token before it moves) or v will be requested befor the fault and the server on v wil not be the least recently used.

Proof of Proposition 18. Let t_0 ($<t_1$) be the last time node v was requested. No nodes in $T(v)$ are requested in the interval from t_0 to t_1, denoted (t_0, t_1). On the other hand, all the nodes in *LRUtree*(t_1) are requested in this interval.

Because all the lonely LRU servers have tokens, at time t_0, the number of OPT servers in $T(v)$ is greater than or equal to the number of LRU servers without tokens in $T(v)$.

No nodes in $T(v)$ are requested in the interval (t_0, t_1), so no node in $T(v)$ can be given a token twice. Thus, the number of LRU servers without a token in $T(v)$ at time t_0 is greater than or equal to the number of tokens placed on nodes in $T(v)$ in the interval (t_0, t_1).

Consider $LRUtree(t)$ for some point in time in the interval (t_0, t_1). $LRUtree(t)$ has a leaf in $T(v)$. Whenever OPT incurs a fault in the interval (t_0, t_1), the first node without a token reached on the path from this leaf to the requested node gets a token. This node is in $T(v)$ because node v is on the path and never gets a token. Thus, the number of tokens that are placed on nodes in $T(v)$ in the interval (t_0, t_1) is greater or equal the number of OPT faults in the interval (t_0, t_1).

We now count the number of OPT faults in the interval (t_0, t_1). Examine the set of nodes that are occupied by lonely LRU servers at time t_1. These nodes have been requested since time t_0 because each lonely LRU server has serviced a request since time t_0. For each one of these nodes, an OPT server has left this node in the interval (t_0, t_1). Furthermore, none of these nodes are in $T(v)$ because no node in $T(v)$ is requested in the interval (t_0, t_1). This yields that the number of OPT faults in the interval (t_0, t_1) is greater than or equal to the number of OPT servers that leave a node in $T(v)$ in the interval (t_0, t_1) plus the number of lonely LRU servers at time t_1.

Putting all the inequalities together: The number of OPT servers in $T(v)$ at time t_0 is greater than or equal to the number of OPT servers that leave $T(v)$ in the interval (t_0, t_1) plus the number of lonely LRU servers at time t_1. Thus the number of OPT servers remaining in $T(v)$ at time t_1 is at least the number of lonely LRU servers at time t_1. Since the number of lonely OPT servers is the same as the number of lonely LRU servers, all the lonely OPT servers are in $T(v)$ at time t_1. ∎

COROLLARY 19. *When G is a path, $c_{\mathrm{LRU},k} = 1$*

We conclude this subsection by considering a mesh. Suppose that G is an $n \times n$ mesh. It is easy to see that G contains a biconnected subgraph with $k+1$ nodes. This implies that $c_{\mathrm{LRU},k} = k$. On the other hand, G contains as a subgraph a tree with $k+1$ nodes that has $\lfloor 2/3(k+1) \rfloor + c\sqrt{k}$ leaves, for some constant c. Hence, $c_k(G) \geq a(G) \geq \lfloor 2/3(k+1) \rfloor + O(\sqrt{k})$. The next proposition follows.

PROPOSITION 20. *When G is a square mesh $c_{\mathrm{LRU},k}(G) \leq 3/2 c_k(G) + o(1)$.*

3.3. *Caching*

LRU is used both as a paging strategy as well as a caching strategy. In caching, the number of blocks in the cache is often quite small (i.e., $k = 2, 4$), so it is of special interest to examine LRU's behavior for these values of k. In these cases, LRU compares very favorably to the performance of any on-line algorithm.

THEOREM 21. *For all G, $c_{\mathrm{LRU},2}(G) = c_2(G)$.*

Proof. If G is a tree, then the optimality of LRU is assured by Theorem 12. So we can now assume G has a cycle of length $l \geq 3$. The following two propositions are clearly sufficient to show the optimality of LRU when $k = 2$.

PROPOSITION 22. *If the smallest cycle in G has length $l \geq 3$, then $c_{\mathrm{LRU},2}(G) \leq 1 + 1/(l-2)$.*

PROPOSITION 23. *If G has a cycle of size $l \geq 3$, then $c_2(G) \geq 1 + 1/(l-2)$.* ∎

Proof of Proposition 22. We will account for the relative costs by considering phases of requests $v_1, v_2, \ldots, v_t$ (for $v_j \neq v_{j+1}$), where LRU and the adversary are aligned before the request to v_1 (say on nodes v_{-1} and v_0) and v_t is the first request on which LRU pays but the adversary does not. Note that after serving v_t, LRU and the adversary must be aligned on nodes v_t and v_{t-1} so that a new phase begins. We need to show that in any phase, the ratio of LRU's cost to the adversary's cost is no more than $1 + 1/(l-2)$. In order to eventually avoid paying for v_t while LRU pays, the adversary must clearly separate its servers. Say the last time this separation takes place in the phase is on the request to node v_i with $1 \leq i < t$ so that the servers are not aligned again until the request to v_t. If the request sequence ever reverses itself after the request to v_i (i.e., $v_{j+1} = v_{j-1}$ for some $j \geq i$), then the adversary must pay to cover v_{j+1} while LRU does not pay so that when the phase ends, LRU and the adversary will have had the same cost in this phase. So we may assume the sequence $v_i, v_{i+1}, \ldots, v_t$ does not reverse itself. Clearly the adversary has left a server on v_{t-2} while serving $v_i, v_{i+1}, \ldots, v_{t-1}$ and $v_t = v_{t-2}$. That is, $v_{i-2}, v_{i-1}, \ldots, v_t$ is a (perhaps not simple) cycle of length $t - i + 2 \geq l$ the adversary pays $t - i$ on $v_i, v_{i+1}, \ldots, v_{t-1}$, while LRU pays $t - i + 1$. Clearly $(t-i+1)/(t-i) \leq 1 + 1/(l-2)$. ∎

Proof of Proposition 23. Let A be any on-line algorithms. We consider a phase to start and end when the adversary and A have their servers aligned. Let $v_0, v_1, \ldots, v_{l-1}, v_0$ be an l-cycle and without loss of generality suppose that the phase begins with servers on nodes v_0 and v_{l-1}. The adversary requests nodes $v_1, v_2, \ldots$.

Case 1. A performs like LRU on $v_1, v_2, \ldots, v_{l-2}$ and does not separate its servers. Then the adversary requests v_{l-1} and serves $v_1, \ldots, v_{l-1}$ at the optimal cost of $l-2$ (having left a server on v_{l-1}). The adversary then continues to request v_{l-2}, v_{l-1} until A aligns its servers on v_{l-1} and v_{l-2}. Clearly A's cost is at least $l-1$.

Case 2. A separates its servers while serving the request for node v_i for some i satisfying $1 \leq i \leq l-2$. At this point, A will have servers on v_i and $v_{(i-2) \bmod l}$ and its cost

on the requests $v_1, ..., v_i$ will be i. On the other hand, the adversary will serve the requests $v_1, ..., v_i$ as would LRU at the same cost i, but with servers now on v_i and v_{i-1}. The adversary then keeps requesting v_{i-1}, v_i until A aligns its servers with the adversary to that A's cost at the end of the phase is at least $i+1$. Clearly $1 + 1/i \geqslant 1 + 1/(l-2)$. ▮

THEOREM 24. *For all* G, $c_{\mathrm{LRU},4}(G) \leqslant (\frac{3}{2})\, c_4(G)$.

Proof. To prove the theorem we use three propositions.

PROPOSITION 25. *If* G *has a biconnected component on five nodes, then* $c_4(G) \geqslant 3$.

PROPOSITION 26. *If* G *has no biconnected subgraph on five nodes, then* $c_{\mathrm{LRU},4}(G) \leqslant 3$.

PROPOSITION 27. *If* G *is an* n*-cycle, for* $n > 5$, *then* $c_{\mathrm{LRU},4}(G) \leqslant (\frac{5}{4})\, c_4(G)$.

Before proving these propositions we show how they imply Theorem 24. We consider four possible cases:

Case 1. The graph G has at least one vertex of degree at least four. Then, $c_4(G) \geqslant 3$ by Proposition 4. Further, for every graph G, $c_{\mathrm{LRU},4}(G) \leqslant 4$.

Case 2. The graph G has a biconnected component of size five. Then, by Proposition 25, $c_4(G) \geqslant 3$.

Case 3. Not cases (1) and (2), and the graph G has at least one vertex of degree three. Then, $c_4(G) \geqslant 2$ by Proposition 4. By Proposition 26, $c_{\mathrm{LRU},4}(G) \leqslant 3$.

Case 4. Not Cases 1, 2, and 3. In this case the graph G is either a path or an n-cycle for some $n > 5$. If G is a path, then $c_{\mathrm{LRU},4}(G) = 1$. Otherwise, by Proposition 27, $c_{\mathrm{LRU},4}(G) \leqslant (\frac{5}{4})\, c_4(G)$. ▮

Proof of Proposition 25. The request sequence is restricted to a five node biconnected subgraph of G. We start with the servers aligned. The adversary requests the vacant node v. Suppose that the on-line algorithm vacates node u to service the request. Because the graph is biconnected, there is a path from v to u that passes through at most one other vertex, w. The adversary requests w then u. On-line vacates node x to service the request to u. Because the graph is biconnected, there is a path from u to x that does not use nodes other than w and v. The adversary takes this path to node x. The on-line algorithm has incurred three faults, while there is still a node z that has not been requested. The off-line algorithm services the initial request to node v with the server on node z, and thus only incurs one fault. The adversary requests nodes v, w, u, and x until the servers are aligned again. ▮

Proof of Proposition 26. Recall that LRU server is *lonely* if the node it occupies is not occupied by an OPT server. Similarly, an OPT server is *lonely* if the node it occupies is not occupied by an LRU server. We prove two claims.

CLAIM 1. *Start with the servers aligned. If within three LRU faults OPT has incurred only one fault and the servers are not aligned, then within three more LRU faults, either OPT incurs two faults or OPT incurs one fault and the servers become aligned.*

CLAIM 2. *From any configuration, within three LRU faults, the servers are aligned or OPT has incurred a fault.*

If the servers are aligned, we start a phase when the servers become unaligned and end the phase when the servers are realigned. The two fact imply that the number of LRU faults in a phase is not more than three times the number of OPT faults in a phase. ▮

Proof of Claim 1. A sequence of nodes (v_1, v_2, v_3, v_4) is said to be *LRU-connected* if it is possible for v_i to be occupied by the ith most recently used server for all $1 \leqslant i \leqslant 4$. In such a case v_1 is adjacent to v_2, v_3 is adjacent to either v_1 or v_2, and v_4 is adjacent to one of $\{v_1, v_2, v_3\}$.

We start with the servers aligned. Let t_1 be the time of the first LRU fault. OPT faults at time t_1 as well. After t_1, there is one lonely LRU server. Let v_o be the node occupied by the lonely OPT server. Let v_i be the node occupied by the ith most recently used server. Thus, $v_1, v_2, v_3, v_4)$ are LRU-connected. By examining the configuration just before t_1, (v_2, v_3, v_4, v_o) are LRU-connected.

If a lonely LRU server services a request, then either OPT incurs a fault or the number of lonely LRU servers decreases by one. Let t_2 and t_3 be the times of the next two LRU faults. Suppose OPT does not fault before time t_3 and the servers are not aligned at time t_3. Then the lonely LRU server has not serviced a request in the interval from t_1 to t_3. This means the server on v_2 is the lonely server and the servers on v_4 and v_3 service the two LRU faults.

After the request at time t_1, the request sequence must go from v_1 to v_o; LRU services the request with the server on v_4. Then the request sequence goes to v_4 and LRU uses the server on v_3 to service the request. Furthermore, each of these nodes is hit without hitting any other nodes, and v_o is reached without hitting v_4. Thus, v_1 is adjacent to v_o and v_4 is adjacent to either v_0 or v_1.

With the above adjacency information, and the fact that (v_1, v_2, v_3, v_4) and (v_2, v_3, v_4, v_o) are LRU-connected, it can be determined that either the subgraph induced by $\{v_1, v_2, v_3, v_4, v_o\}$ is biconnected or the subgraph induced by $\{v_1, v_2, v_4, v_o\}$ is biconnected. Since there are no five node biconnected subgraphs in G, we can assume the latter. The edges (v_o, v_1), (v_1, v_2), (v_2, v_3) are in the graph. Also, v_4 is adjacent to v_o or v_1 and to v_2 or v_3. v_o is adjacent to one of $\{v_2, v_3, v_4\}$.

Any node can be connected to at most one node in $\{v_1, v_2, v_4, v_o\}$. Thus, v_3 is adjacent only to v_2 (since (v_2, v_3, v_4, v_o) are LRU-connected).

After time t_3, v_2 is still occupied by the lonely LRU server

which is now the least recently used. v_3 has the lonely OPT server and the current request is one of $\{v_o, v_1, v_4\}$. Witout loss of generality, suppose it is v_1. In order to make only LRU fault, a new node must be requested (v_3 is only reachable through v_2). Call the new node v_n. If the servers are unaligned at his point, then v_3 still contains the lonely OPT server. In order to make only LRU fault, v_3 must be requested.

After time t_3, the sequence goes from v_1 to v_n to v_3. LRU faults twice and OPT faults once. We prove that if there is one more LRU fault and OPT does not fault, then the severs become aligned. The lonely OPT server occupies one of $\{v_o, v_4\}$. In order to hit the lonely OPT server, the sequence must go back through v_n and v_1 (v_n and v_3 are connected to only one node in $\{v_1, v_2, v_o, v_4\}$. v_n is adjacent to v_1 and v_3 is adjacent to v_2). Thus $\{v_3, v_n, v_1\}$ are occupied both by LRU and OPT servers and the lonely LRU server is the least recently used. If on the next LRU fault, OPT does not fault, then the servers become realigned. ∎

Proof of Claim 2. If a lonely LRU server moves, the number of lonely LRU servers decreases by one or OPT incurs a fault. If a server becomes lonely then OPT incurs a fault. All the LRU servers that are currently lonely will service a request within the next three LRU faults. If OPT does not incur a cost then after three LRU faults the servers will be realigned. ∎

Proof of Proposition 27. Observe that for $n > 5$, $c_{\mathrm{LRU},4}(n\text{-cycle}) \leqslant (n-1)/(n-4)$. We now show that $c_4(n\text{-cycle}) \geqslant (n-2)/(n-4)$. The proposition follows.

Assume the on-line and off-line servers are aligned on nodes $\{1, 2, 3, 4\}$ and that $\{5, 6, ..., n\}$ are vacant. Suppose that the adversary requests node 5. There are two cases.

Case 1. If the on-line algorithm does not vacate node 1 (say leaving an on-line hole at $i \in \{2, 3, 4\}$), then the adversary requests 4, ..., i and then forces a ratio $\geqslant 2$ by the adversary having served 5 with the server at 1 and then forcing realignment at $\{2, 3, 4, 5\}$.

Case 2. The on-line has responded to request for 5 by vacating node 1 and now has servers on $\{2, 3, 4, 5\}$. The adversary now requests node 6. There are two subcases.

Case 2.1. The on-line algorithm responds to the request by vacating node $i \in \{2, 3\}$. The adversary then requests 7, ..., n, 1, 2, 3 and forces realignment on nodes $\{n, 1, 2, 3\}$. The total cost to the on-line algorithm is at least two (for the requests to $\{5, 6\}$) $+ n - 4$ (for the requests to $\{7, ..., n, 1, ..., i\}$) $= n - 2$. The adversary pays $n - 4$ by serving all faults with the server originally on node 4.

Case 2.2. The on-line algorithm responds to the request by vacating node $i \in \{4, 5\}$. The adversary then requests 5, 4, 5, 6, ..., n, 1 and forces realignment on $\{n-2, n-1, n, 1\}$. The total cost to the on-line algorithm is at least three (for the requests to $\{5, 6, 5, 4, 5, 6\} + n - 5$ (for the requests to $\{7, ..., n, 1\} = n - 2$. The adversary pays $n - 4$ by serving the faults at $\{5, 6\}$ with the servers originally on nodes $\{2, 3\}$ and then alternating these same two servers in an LRU fashion so as to serve the faults at $\{7, ..., n\}$ and finish the request sequence with servers on $\{n-2, n-1, n, 1\}$. ∎

3.4. *Comparison of LRU vs FIFO*

We use access graphs to differentiate between the competitiveness LRU and FIFO can achieve. We have determined LRU's competitiveness on any graph up to a constant factor. Now, we need to lower bound FIFO's competitiveness.

THEOREM 28. *For any* G *with* $n \geqslant k+1$, $c_{\mathrm{FIFO},k} \geqslant (k+1)/2$.

Proof. The adversary picks a subgraph H on $k+1$ nodes and only requests nodes from that subgraph. Let v and w be two adjacent nodes in H such that w is not an articulation node. At the beginning of each phase FIFO has node v vacant and the server on node w is the server that has moved the most recently. On a fault, FIFO moves the server that has moved the least recently. The adversary requests every node where FIFO has its hole until every node (except v) has been vacated once. The paths from hole to hole in the access graph are covered by FIFO's servers. Requesting these nodes does not effect FIFO's behavior. Furthermore, since w is not an articulation node, the adversary can move from hole to hole without requesting w. At this point, every one of FIFO's servers has moved one. Note w is vacant, and the server that occupies node v has moved the least recently. The adversary requests node w and FIFO moves the server on node v to node w. The phase ends: the hole is on node v, an the server on node w is the server that FIFO moved most recently. During the phase, FIFO incurs a fault on every node in the graph ($k+1$ faults). The adversary services the sequence by moving the server on node w to node v and then back to node w for the last request in the phase (2 faults). ∎

Theorem 28 and the fact that LRU is k-competitive on any G [22] imply that $c_{\mathrm{LRU},k}(G) \leqslant 2c_{\mathrm{FIFO},k}(G)$ for all G. Theorems 9, 13, 21, and 24 show that on many graphs, however, $c_{\mathrm{LRU},k}(G)$ is much smaller than $(k+1)/2$. This

provides an explanation of why LRU should be preferred to FIFO in practice.

4. NEARLY OPTIMAL ALGORITHMS FOR PAGING

Is there an on-line paging algorithm whose competitiveness is close to $c_k(G)$ on every graph G? We seek a "universal" algorithm—informally, an algorithm whose description is independent of G (and hopefully succinct). LRU and FIFO are universal algorithms, but by Example 2 neither is close to optimal on all G. We now describe a simple and natural algorithm FAR and show that its competitiveness is close to $c_k(G)$ on every G.

FAR is a marking algorithm. Consider a phase of FAR. Call the nodes requested (and marked) in the current phase *red* and the nodes requested (and marked) in the previous phase *blue*. Note that all blue nodes have servers on them at the beginning of the current phase.

FAR chooses the unmarked server to use on a fault by vacating a blue node whose distance to the nearest red node (in G) is a maximum. The intuition behind FAR is as follows: it is known [1, 17, 18] that the optimal (off-line) paging algorithm on any sequence is to vacate the node whose next request occurs furthest in the future. FAR attempts to approximate this by vacating a node that is far from currently red nodes in G and, thus, likely to be requested far in the future.

THEOREM 29. *For any G and k, $c_{\mathrm{FAR},k}(G) \leqslant 2 + 4c_k(G)\lceil \log 2k \rceil$.*

In words, $c_{\mathrm{FAR},k}(G)$ is within $2 + 4\lceil \log 2k \rceil$ of $c_k(G)$ on every G, a performance LRU and FIFO cannot match. Bye Proposition 6, we know that $C_{\mathrm{FAR},k}(G) \leqslant k$, implying that FAR is optimal in the Sleator–Tarjan model (G is the complete graph), including the constant. Recently, Irani *et al.* [10] have shown that on every undirected access graph G, $c_{\mathrm{FAR},k}(G)$ is within a constant multiple of $c_k(G)$, thus improving on Theorem 29 above and showing that the performance of FAR cannot be improved by more than a constant factor. On the other hand, there are graphs G, where $C_{\mathrm{FAR},k}(G) \geqslant \theta C_k(G)$, for some constant $\theta > 1$.

Proof of Theorem 29. Let a *new* node be a node requested in the current phase that is not blue. Let g be the number of new nodes in the current phase. By Proposition 6, the adversary incurs at least $g/2$ faults for this phase, amortized. We bound the number of faults in the phase for FAR by the number of nodes vacated by FAR in the phase.

Let $M_G = \max_{T \in \mathcal{T}_k(G)} |I(T)|$, and let D_G be the maximum diameter of any connected k-node induced subgraph of G. We divide the current phase into at most $1 + \lceil \log D_G \rceil$ *sub-phases*, showing that at most $2gM_G$ blue nodes are vacated during each sub-phase. This will yield a bound of at most $g + 2gM_G(1 + \lceil \log D_G \rceil)$ faults in the phase, which is at most $2 + 4c_k(G)\lceil \log 2k \rceil$ times the number of faults of the adversary in the phase. (The extra g faults are on new nodes.)

Let $v_1, v_2, \ldots$ be the sequence of nodes vacated in the phase, and let d_i be the distance from v_i to the nearest red node at the time it is vacated. The sequence $d_1, d_2, \ldots$ is non-increasing, and $d_1 \leqslant D_G$. Let i_2 be the smallest index such that $d_{i_2} \leqslant d_1/2$; the first sub-phase ends with the vacation of v_{i_2-1} and the second sub-phase begins with v_{i_2}. In general, the jth sub-phase, for $j \geqslant 2$, begins at the smallest index i_j such that $d_{i_j} \leqslant d_{i_{j-1}}/2$. Let S_j be the set of red nodes at the beginning of sub-phase j.

Divide the sequence of nodes vacated in the jth sub-phase into *blocks* of g successive requests. Because the number of vacant blue nodes at any time is at most g, at least one node of each block is marked by the time *any* node of any subsequent block is vacated. Pick such a node for each block, and call it a *rep*. We bound the number of reps by $2M_G$, and this will imply a bound of $2gM_G$ on the number of nodes vacated in the sub-phase. Let r_j be the number of reps in sub-phase j. The number of nodes marked during the subphase is at least $r_j f_{i_j}/2$. This follows from the fact that when a rep is marked, the next rep to be marked is at distance of at least $d_{i_j}/2$ from the set of marked nodes. Also, at the beginning of the sub-phase the first rep to be marked is at a distance of at least $d_{i_j}/2$ from the set of marked nodes. Thus, at least $d_{i_j}/2$ new nodes are marked between suggessive reps being marked.

We get that for any two reps u, v, the distance from u and v to any member of S_j is greater than $d_{i_j}/2$. Also, the distance between u and v is greater than $d_{i_j}/2$. (The last follows from the fact that the second of $\{u, v\}$ to be vacated must be at distance greater than $d_{i_j}/2$ from the first, which is in the set of marked nodes by then.) The shortest path then from any rep to S_j is at most d_{i_j} and does not contain any other reps.

Imagine contradicting S_j to a node; we now argue that a tree can be grown from S_j whose leaves include at least half of the reps. Furthermore, the size of the tree is no larger than $|S_{j+1} - S_j|$. Thus if we expand S_j and take a minimum spanning tree of S_j, the size of the expanded tree is no larger than $|S_{j+1}|$ which is no more than k. To build the tree, pick half of the reps. Consider the union of the shortest paths of each rep to S_j. The set of nodes contained in the union has no more than $r_j d_{i_j}/2$ nodes. Because none of the paths contain any reps, each of the reps is a leaf of a spanning tree of the set. ∎

We note that the bound cannot be improved by much in terms of M_G and D_G: it is tight to within a constant factor on both the complete graph and the $(k+1)$-node cycle. The recent work of Irani *et al.* [10] shows that FAR achieves essentially the best possible competitiveness on every access graph G. Their proof makes use of the vine decomposition lower bound introduced in Section 2, thus establishing additionally that the vine decomposition provides the best

possible lower bound (to within a constant) on $c_k(G)$ for every access graph G.

For the special case $n = k + 1$ we present another algorithm, which we call two-second (TS), that does achieve a constant factor from optimal.

We describe TS using the same terminology used to describe FAR. TS is a marking algorithm and proceeds in phases. Each phase of TS consists of two sub-phases. In the first sub-phase TS serves each request by vacating a blue node whose degree is not two, if such a node exists. (The specific node is selected arbitrarily.) The sub-phase ends when all such nodes are red. At the end of this sub-phase all remaining blue nodes are of degree two and hence can be partitioned (in a unique way) into node disjoint paths connecting nodes of degree other than two. In the second sub-phase TS serves each request by vacating the middle node of one of the current blue paths (chosen arbitrarily). The second sub-phase ends when k nodes are red. For $i = 1, 2$, let r_i be the number of faults in te ith sub-phase.

There exists a tree T such that $r_1 = O(|l(T)|)$. This follows from the following graph-theoretic proposition [13].

PROPOSITION 30. *Let $n_2(G)$ be the number of nodes of degree $\neq$ in a graph G. Then, there exists a tree $T \subseteq G$ such that $|l(T)| = \Omega(n_2(G))$.*

PROPOSITION 31. *Let $\mathscr{P}$ be the set of paths consisting of the unmarked blue nodes at the end of the first sub-phase. Then $r_2 = O(\sum_{P \in \mathscr{P}} 1 + \lfloor \log |P| \rfloor)$.*

Proof. Each request is served by the server at the mid-point of some path $P \in \mathscr{P}$. The next fault is when this mid-point is requested. However, to request this node at least half of the nodes on P have to be marked. The bound follows. ∎

To prove that $r_1 + r_2$ is within a constant factor of optimal we use the lower bound given by the vine decompositions of G. Taking the tree T of Proposition 30 as a tree in a vine decomposition of G, there exists a vine decomposition $(T, \mathscr{P})$ that gives a lower bound of at least r_1. Taking another vine decomposition whose set of paths $\mathscr{P}$ is the set of blue paths at the end of the first sub-phase, there exists a vine decomposition $(T, \mathscr{P})$ that gives a lower bound of at least r_2. Since the lower bound is given by the maximum over all vine decompositions $c_k(G)$ is $\Omega(r_1 + r_2)$.

THEOREM 32. *For any G and $k = n - 1$, $c_{\mathrm{TS}.k}(G)$ is $O(c_k(G))$.*

Again, since TS is a marking algorithm, we also have $c_{\mathrm{TS}.k}(G) \leqslant k$.

5. EXTENSIONS AND FURTHER WORK

5.1. *Directed Graphs and Structured Program Graphs*

We now consider a class of directed graphs that captures the control flow of a program (Example 3). A *structured program graph* is meant to model the access pattern generated by an execution of a program written in a structured programming language. It is a directed graph built from the following rules:

(i) There is a unique source node s and a unique sink t, and a directed path from s to t.

(ii) A structured program graph can be derived from another by attaching to a node v in it a stricted cycle $\mathscr{C}$: we identify v and one node in $\mathscr{C}$. This is to model loops in a program (DO/WHILE loops).

(iii) Series/parallel composition. Two program graphs can be composed in parallel by identifying their sources and sinks, respectively, to form a new program graph. This is to model branching (IF/CASE statements). Two program graphs can be composed in series: the sink of one is identified with the source of the other.

Note that an arbitary GOTO statement cannot be captured by such a graph. We now present a variant of FAR which we call 2FAR, and prove that $c_{\mathrm{2FAR}.k}(G)$ is within a constant factor of $c_k(G)$ for any structured program graph G, provided every strongly connected component of G has size at most $k + O(1)$.

2FAR is also a marking algorithm; we describe it here for the case when every strongly connected component of G has size at most $k + 1$, showing that it is optimal in this case. On a fault at node v, 2FAR decides which server at an unmarked node to choose as follows. If there is a node u not reachable from v that has a server on it, it uses the server at u. If not, let H be the maximal strongly connected component containing v, comprised of cycles $C_1, \ldots, C_m$ used in its inductive construction by rule (ii) above. Form an undirected graph D each of whose nodes represents a cycle C_i, with an edge between two nodes if the corresponding cycles share a node. Consider the sub-graph F of D induced by those C_i that currently contain any blue nodes (borrowing the terminology of Section 4). Rather than choose the server on the blue node at the greatest distance from the set of currently marked nodes (as FAR would), 2FAR uses a two-stage process. Call a node in F a *peripheral node* if it is not an articulation node in F. It first selects (arbitrarily) a peripheral node C_i in F; this is the cycle from which it will bring the server. Next, let u be the node of C_i linking it to the rest of F; the server that is furthest from the marked node is used. Notice that on a $k + 1$ node cycle, 2FAR is exactly MRU (most recently used).

THEOREM 33. *For any structured program graph G in which every strongly connected component has size $\leqslant k + 1$, $c_{\mathrm{2FAR}.k}(G) = c_k(G)$.*

Proof. Since 2FAR always first tries to use a server from a portion of the access graph that is unreachable from the present request, we confine our attention to the strongly

connected components of G. Let us focus on such a component C containing $k+1$ nodes, k of which have servers on them (call the unoccupied node the *hole*). Note that when the hole is on one of the loops in C, every node in that loop is requested before 2FAR incurs a fault on that hole. This leads us to view each loop in C as a "super-node" that is requested all at once in the request sequence. To this end we represent C as a tree T in the following manner. These are two types of nodes in T, *loop nodes* and *cut nodes*. There is one loop node representing each loop of C. There is one cut node representing each node of C to which a loop has been attached by rule (ii) above in the synthesis of structured program graphs. An edge in T exists between a loop node and a cut node if the corresponding loop contains the corresponding node. Thus each leaf of T is a loop node. At any time the hole resides in a loop of C, and we think of it as residing at the corresponding loop node of T.

2FAR always keeps the hole node that is a leaf of T, moving it to a new leaf of T on incurring a fault. Let l be the number of leaves of T. It is easy to see now (along the lines of Proposition 4) that the competitiveness of any deterministic on-line algorithm is at least $l-1$, and 2FAR achieves this bound. ∎

Note that LRU cannot achieve this performance (because of the $(k+1)$-node cycle). It is possible to extend the above proof to show that on structured program graphs in which no strongly connected component has more than $k+g$ nodes, $c_{\mathrm{2FAR},k}(G)$ is at most $gc_k(G)$. This is because 2FAR may incur g faults moving the hole from one leaf to another. However, we suspect that the actual performance of 2FAR is better. Irani *et al.* [10] have recently shown that a natural extension of 2FAR achieves a competitiveness within a constant multiple of $c_k(G)$ for all structured program graphs.

5.2. *Randomized Paging Algorithms*

We return to undirected access graphs. It has ben shown [6, 18] that in the Sleator–Tarjan model, the competitiveness of the optimal randomized on-line paging algorithm is H_k (the kth harmonic number) agains an oblivious adversary [2]. Can one always hope for such a dramatic improvement over deterministic algorithms? We show that when G is the $k+1$ node circle, no randomized algorithm can achieve a competitiveness better than $(\lceil \log k \rceil)/2$. Thus randomization can help by at best a constant factor in this case. We then show that a variant of the marking algorithm in [6] is within a constant factor of the optimal randomized algorithm for the important case when G is a tree.

PROPOSITION 34. *No randomized paging algorithm achieves a competitiveness less than* $\lceil \log(k+1) \rceil/2$ *on the* $(k+1)$*-node cicle against an oblivious adversary.*

Proof. By the von Neuman minimax principle [25], it suffices to provide a probability distribution on input sequences on which the expected number of faults for any deterministic on-line algorithm is at least $\lceil \log(k+1) \rceil/2$ during a period when an off-line algorithm makes only one fault. The sequence is generated in phases, each of which ends when every node has been hit at least once in the phase (so that the off-line algorithm incurs one fault in each phase). At the beginning of a phase, the sequence first walks from its present position to the point diametrically across the circle, choosing one of the two ways of getting there equiprobably. From there, it proceeds to the mid-point of the nodes yet to be hit in the phase, choosing equiprobably one of the two ways of getting there, and so on. Thus on each of these $\lceil \log(k+1) \rceil/2$ steps it causes the on-line algorithm to fault with probability $\frac{1}{2}$. ∎

Let $M = \max_{t \in T_{k+1}(G)} |l(t)|$. The proof of the following theorem follows from [6].

THEOREMM 35. *For any randomized algorithm* R, $c_{R,k}(G) \geq H_{M-1}$.

Consider the following randomized marking algorithm for tree access graphs, which is a variant of the algorithm in [6]. The algorithm proceeds in phases, as in all marking algorithms. Let T be the tree formed by the set of nodes marked in the previous phase. Call a leaf L of T *dead* when the path from L to the set of nodes marked in this phase has no servers on it. On a fault, pick a live leaf L at random and evict the first server on the path from L to the marked nodes.

THEOREM 36. *There is a constant* b *such that the above randomized algorithm achieves a competitiveness of* $b \log M$ *on every tree access graph* G, *where* $M = \max_{t \in T_{k+1}(G)} |l(t)|$.

Proof. Let g be the number of new nodes requested in a phase. We count faults in T during the phase (there are only g other faults). When a leaf dies, it remains dead for the rest of the phase. Thus the number of live leaves is nonincreasing with time. Label each evicted node with the name of the random (live) leaf that led to its eviction. Every hole in T is labelled with the name of some leaf. After a leaf dies, no more holes are labelled with the name of that leaf.

If l is the number of live leaves at the time a hole is labelled, then the hole is labelled with the name of any particular leaf with probability $1/l$. Suppose that just before a leaf L dies, the number of live leaves is l. Then the expected number of holes labelled L is bounded above by g/l: There are at most g holes and each one is labelled L with probability at most $1/l$.

Let L_i be the ith leaf in T that dies. The number of faults the algorithm incurs on nodes in T is at most the number of nodes in T that get labelled in the phase $\leq \sum_{i=1} Mg/(M-i+1)$, which is at most a constant times $g \log M$. ∎

5.3. *Open Problems*

By Theorem 28, LRU is never more than twice as bad as FIFO. Theorems 9, 13, 21, and 24 show that LRU is often better. It would be worth removing the factor of two:

Open Question 1. Show that for all G and k, $c_{\mathrm{LRU},k}(G) \leqslant c_{\mathrm{FIFO},k}(G)$.

We believe that 2FAR performs far better than our results suggest:

Open Question 2. For directed graphs, how close to $c_k(G)$ is $c_{\mathrm{2FAR},k}(G)$? Is there a near-optimal algorithm for all directed access graphs?

The solution of the following questions appear to need a randomized variant of the two-person game used in proving the vine-decomposition lower bound (Theorem 5):

Open Question 3. Is there a "universal" randomized algorithm that is close to optimal on every G? Given G, what is a lower bound on the competitiveness of a randomized algorithm agains an oblivious adversary?

ACKNOWLEDGMENTS

We thank Don Coppersmith, Amos Fiat, Anna Karlin, Howard Karloff, Hugh Thomas, and the referees for their comments and suggestions.

REFERENCES

1. L. A. Belady, A study of replacement algorithms for virtual storage computers, *IBM Systems J.* **5** (1966), 78–101.
2. S. Ben-David, A. Borodin, R. M. Karp, G. Tardos, and A. Wigderson, On the power of randomization in on-line algorithms, *in* "Proceedings, 22nd Annual ACM Symposium on Theory of Computing, 1990," pp. 379–388.
3. A. Borodin, N. Linial, and M. Saks, An optimal on-line algorithm for metrical task systems, *in* "Nineteenth Annual ACM Symposium on Theory of Computing, 1987," pp. 373–382.
4. P. J. Denning, Working sets past and present, *IEEE Trans. Sofware Eng.* **SE-6** (1980), 64–84.
5. D. Ferrari, The improvement of program behavior, *IEEE Comput.* **9** (1976), 39–47.
6. A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young, On competitive algorithms for paging problems, *J. Algorithms* **12** (1991), 685–699.
7. P. A. Franaszek and T. J. Wagner, Some distribution-free aspects of paging performance, *J. Assoc. Comput. Mach.* **21** (1974), 31–39.
8. D. Hatfield and J. Gerald, Program restructuring for virtual memory, *IBM J. Systems Technol.* **10** (1971), 168–192.
9. W. C. Hobart, Jr. and H. G. Cragon, Locality characteristics of symbolic programs, *in* "IEEE International Conference on Computer Design: VLSI in Computers and Processors, 1989," pp. 508–511.
10. S. S. Irani, A. R. Karlin, and S. J. Phillips, Further results on paging with locality of reference, *in* "Third Annual ACM-SIAM Symposium on Discrete Algorithms, 1992."
11. A. R. Karlin, M. S. Manasse, L. Rudolph, and D. D. Sleator, Competitive snoopy caching, *Algorithmica* **3**, No. 1 (1988), 70–119.
12. T. Kilburn, D. B. G. Edwards, M. J. Lanigan, and F. H. Sumner, One-level storage system, *IRE Trans. Electron. Comput.* **37** (1962), 223–235.
13. D. J. Kleitman and D. B. West, Spanning trees with many leaves, *SIAM J. Discrete Math.* **4**, No. 1 (1991), 99–106.
14. A. Lempel, S. Even, and I. Cederbaum, An algorithm for planarity testing of graphs, *in* "Proceedings, Int. Symp. on Theory of Graphs" (P. Rosenstiehl, Ed.), pp. 215–232, Gordon & Breach, New York, 1967.
15. P. A. W. Lewis and G. S. Shedler, Empirically derived models for sequences of page exceptions, *IBM J. Res. Develop.* **17** (1973), 86–100.
16. M. S. Manasse, L. A. McGeoch, and D. D. Sleator, Competitive algorithms for on-line problems, *J. Algorithms* (1990), 208–230.
17. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems J.* **9**, No. 2 (1970).
18. L. A. McGeoch and D. D. Sleator, "A Strongly Competitive Randomized Paging Algorithms," Technical Report CMU-CS-89-122, Carnegie–Mellon University, Pittsburgh, PA, 1989; *Algorithmica*, submitted.
19. L. A. McGeoch, D. D. Sleator, and C. Tomasi, Decision procedures for competitive algorithms, in preparation.
20. P. Raghavan and M. Snir, Memory versus randomization in on-line algorithms, *in* "16th International Colloquium on Automata, Languages, and Programming," μ Lecture Notes in Computer Science, Vol. 372, pp. 687–703, Springer-Verlag, New York/Berlin, 1989; revised version abailable as IBM Research Report RC15840, June 1990.
21. G. S. Shedler and C. Tung, Locality in page reference strings, *SIAM J. Comput.* **1** (1972), 218–241.
22. D. D. Sleator and R. E. Tarjan, Amortized efficiency of list update and paging rules, *Comm. ACM* **28** (1985), 202–208.
23. J. R. Spirn, "Program Behavior: Models and Measurements," Elsevier Comput. Sci. Lib., Elsevier, Amsterdam, 1977.
24. H. Whitney, Non-separable and planar graphs, *Trans. Amer. Math. Soc.* **34** (1932), 339–362.
25. A. C-C. Yao, Probabilistic computations: Towards a unified measure of complexity, *in* "17th Annual Symposium on Foundations of Computer Science, 1977," pp. 222–227.
26. N. Young, Competitive pagig as cache-size varies, *in* "Proceedings, Second ACM-SIAM Symposium on Discrete Algorithms, 1991."