Competitive Paging with Locality of Reference

(Preliminary Version)

Allan Borodin [*]
Sandy Irani [‡]
Prabhakar Raghavan [†]
Baruch Schieber [†]

Abstract

The Sleator-Tarjan competitive analysis of paging [19] 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 competitiveness of LRU is much larger than observed in practice. In addition, 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.

[*]Department of Computer Science, University of Toronto. 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.

[†]IBM T.J. Watson Research Center, Yorktown Heights, NY.

[‡]Computer Science Division, UC Berkeley. 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.

1. Overview

This paper deals with the competitive analysis of paging algorithms [10, 19]. 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 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 is evicted to make room for it. An algorithm is *on-line* if 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 to that of the optimal algorithm for the case where the input sequence is drawn from an arbitrary probability distribution, in the spirit of competitiveness. Under a "worst-case" input any paging algorithm can be made to fault on every request; Sleator and Tarjan [19] proposed the alternative of amortized analysis: let $A(\sigma)$ be the number of faults made by a paging algorithm A on request sequence σ, and let $OPT(\sigma)$ be the number of faults made by an optimal algorithm (that has the entire sequence σ available to it in advance) [1]. We say that A is a *c-competitive algorithm* [10] if for all σ, $A(\sigma) - c \cdot OPT(\sigma)$ 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.

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 paging algorithm can achieve a competitiveness less than k, and that a number of algorithms used in practice (including *Least Recently Used* or LRU and *First-In First-Out* or 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 [21] 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, 11, 18] that each time a page is referenced by a program, the next page to be referenced is very likely to come from the same 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. An adversary selects the sequence of page references, subject to 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. All the algorithms we study are *lazy* [14, 17]: a page is evicted only on some fault. We may thus assume that following a request to u, the next request is to an adjacent node. Thus a sequence σ is a walk in the access graph. Sections 2–4 deal with the case when G is undirected, and Section 5.1 deals with directed access graphs. The definition of the competitiveness of an algorithm remains unchanged. 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 $\min_A c_{A,k}(G)$ (the min 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* [14]. However, note that we are not considering the k-server problem [14] 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. 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 that might arise in practice, and then describe related prior work. Let $|V| = n$.

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.*

Example 2: *Let G be a cycle on $n = k + 1$ nodes. It is easy to see that $c_{LRU,k}(G)$ and $c_{FIFO,k}(G)$ 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, $c_k(G) = \lceil \log(k + 1) \rceil$. 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 observed in practice — LRU and FIFO perform poorly on loops just larger than k, leading the designers of the ATLAS computer [11] to develop a paging algorithm with a "loop detector"). 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. Denning [4] (and references therein) develops the *working set* model of program behavior for capturing locality of reference. Spirn [20] gives a comprehensive survey of models for program behavior, although competitive results had not been studied at that time. Shedler and Tung [18] were the first to propose a Markovian model of locality of reference: a Markov chain whose states are the pages, with transition probabilities capturing locality. 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, 13], 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 $c_{LRU,k}(G)$)? (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 candidates. (5) What is the power of randomization in the access graph model?

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 in *PSPACE*. 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 6 and 7) determine $c_{LRU,k}(G)$ for every G and k to within a constant factor. 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 typically 2 or 4.

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 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 [14, 17, 19] 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 $n = k + 1$. We believe that both of these algorithms perform better than our present theorems suggest.

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 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

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 a mapping $V \times V^k \times S \rightarrow V^k \times S$ satisfying the condition that $A(u, < v_1, ..., v_k >, s) = (< v'_1, ...v'_k >, s')$ 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. A request sequence $\sigma = u_1, u_2, ...$ is admissible if for all i, $u_i = u_{i+1}$ or $< u_i, u_{i+1} > \in E$ in which case we can extend the mapping to request sequences in the obvious way. Before developing the tools we will need to analyze specific paging algorithms, it would be helpful 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.* [16] for server problems:

Proposition 1: *(1) In general, it is undecidable if a given paging algorithm A achieves a given competitive ratio. (2) For any k, any finite access graph $G = (V, E)$ and any finite state algorithm (i.e., $|S|$ is bounded by a computable function of $n = |V|$), we can compute $c_{A,k}(G)$ in PSPACE(n). (All the paging algorithms we consider are indeed finite state.) (3) For any k, and any finite access graph G, we can compute $c_k(G)$ and furthermore we can construct (in PSPACE(n)) a finite-state on-line algorithm B which realizes this competitive ratio.*

Proof:

1. 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.

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

3. Using the observations of Manasse *et al.* [14] concerning p-residues and the fact that k is an obvious upper bound on $c_k(G)$ for any G, we can restrict attention to algorithms whose state set is a subset of $[-(k+1), (2k+1)]^n$. Again, following [14], 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 $n^k\ (3k+2)^n$ requests, we construct the algorithm B which minimizes the ratio between B and the off-line dynamic programming optimal over all admissible request sequences that cause B to cycle when started in a reachble configuration/state.

To see that these computations are all in *PSPACE(n)*, we note the following:

i) $V^k \times S$ is exponentially bounded.

ii) All reachability problems in a graph with exponentially many nodes can be computed in *PSPACE.* In particular, all simple paths or cycles from a given node can be enumerated in *PSPACE.*

iii) For a given request sequence σ of at most exponential length, the cost of an on-line algorithm B (and of the optimal off-line algorithm σ) can be computed in *PSPACE.*

□

Observe that $c_{A,k}(G)$ and $c_k(G)$ are monotone in the addition of edges to G; thus $c_{A,k}(G)$ is bounded above by the competitiveness of A in the Sleator-Tarjan model (G is the complete graph). 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 $\ell(T)$ denote the set of leaves of T. From the above observation and from ideas similar to those in [19] we have:

Proposition 2: *For any G, k, and any on-line algorithm A, $c_{A,k}(G) \geq \max_{T \in \mathcal{T}_{k+1}(G)} |\ell(T)| - 1$.*

The lower bound of Proposition 2 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 $\ell(T)$ without passing through any other leaf, so that an argument similar to that in [19] 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. 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 $\mathcal{P} = P_1, P_2, \ldots$ of node disjoint paths such that (i) each end-point of each of the paths in adjacent to a node in T; (ii) the nodes of G are partitioned between T and the P_i. Assign to each end-point of each P_i a neighbor in T, and let n_T be the number of leaves of T not assigned to any path. The *value* of a path P, denoted $v(P)$, is defined to be $1 + \lfloor \log |P| \rfloor$. Let $\mathcal{H}_x(G)$ denote the set of connected node-induced subgraphs of G containing x nodes.

Theorem 3:
$$c_k(G) \geq \max_{H \in \mathcal{H}_{k+1}(G)} \{\max_{\mathcal{V}(H)} (\sum_i v(P_i) + n_T)\} - 1$$

Proof Sketch: Here we need only consider the case when G has $n = k + 1$ nodes. In this case, the on-line algorithm can be assumed to always leave one node of G uncovered — the "hole". Following a fault at the hole, the on-line algorithm moves the hole to some other node of the graph. The adversary must, in order to force the next fault, now walk on the graph to the new position of the hole. In the process, it places requests at all the nodes it walks over. During any subsequence in which the adversary has walked over all $k+1$ nodes, it must incur at least one fault for the off-line algorithm. The question then is — how many times must the on-line algorithm move its hole before the adversary is forced to walk on all the $n = k+1$ nodes of the graph (incurring a fault)?

Given a vine decomposition $\mathcal{V}(H) = (T, \mathcal{P})$, we argue that the number of such moves is at least $(\sum_i v(P_i) + n_T) - 1$. When the on-line algorithm moves its hole to a new position following a fault, the adversary sequence walks to that position using one of the following rules: (1) if the new position happens to be a node of T, then the sequence walks along T to that point; (2) if the new position is on a path P, the sequnece walks to the end-point of P closest to T, then along P to the hole. The hole can be reached (causing a fault) at least n_T times by the first rule, and at least $v(P) = 1 + \lfloor \log |P| \rfloor$ using the second rule for each path P in $\mathcal{V}(G)$. □

Many of the algorithms studied in this paper are *marking* algorithms [6, 10]. A marking algorithm proceeds in phases. At the beginning of a phase all the servers are unmarked. Whenever a node is requested, it is marked *red*. On a fault, it vacates a server from an unmarked node (chosen by a rule to be specified) and brings it to the request. A phase ends at the first fault after every server is on a red node; at this point all the nodes become unmarked and a new phase begins. Call the nodes requested (and marked) in the previous phase *blue* nodes, and the nodes requested in this phase but not in the previous phase *new* nodes. Thus all blue nodes have servers on them at the beginning of the current phase. A blue node that is vacated during this phase (the server on it is moved to a fault at another node) becomes *white* (and may eventually become red). We use the following proposition to analyze the competitve ratio of marking algorithms.

Proposition 4: *(a) If g_i new nodes are requested in the ith 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, $c_{A,k}(G) \leq k$.*

The proof of part (a) is due to Fiat *et al.* [6]. The proof of part (b) is due to Karlin *et al.* [10].

The following result of Belady [1] (see also [15] and [20]) 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 that item in fast memory the next access to which occurs furthest in the future in σ. This algorithm is called OPT, and the reason is evident from the following proposition:

Proposition 5: *For every request sequence σ, the cost of algorithm OPT is the same as the optimal cost.*

3. Analysis of LRU and comparison with FIFO

LRU has the property that it maintains all of its servers in one contiguous subgraph of the access graph at all times. This suggests that in analyzing the competitve ratio of LRU, one need only consider small subgraphs of the entire access graph. In fact, the worst case sequence (up to a constant factor) for any access graph can be achieved by requests that are restricted to only a small subgraph. Let $\alpha(H)$ denote the number of articulation nodes in H. (An articulation node is one whose removal disconnects the graph.) Let

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

Theorem 6: $c_{LRU,k}(G) \geq a(G)$.

The proof is complicated and omitted. The idea is to show that the adversary need only request nodes in the subgraph H that achieves the maximum for $a(G)$. We construct a sequence of requests for which LRU faults on every node that is not a articulation node in H except one, while the adversary only incurs a cost of g. The base case for this construction is when $g = 1$, and $\alpha(H) = 0$. In this case we fix two adjacent nodes in H, say s and t. Then, using a numbering of the nodes, similar to st-numbering [12], we construct the sequence of requests.

Theorem 7: $c_{LRU,k}(G) \leq 8a(G)$

Proof Sketch: Break the sequence up 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 4, the cost of the optimal 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. Let f be the number of faults that LRU incurs during the current phase. We want to prove that $f/g \leq 4a(G)$. There are two cases:
Case 1: H has a biconnected subgraph on $\geq k+1$

nodes. Suppose the size of this subgraph is $k + g'$. Then $f/g \leq k/g \leq k/g' \leq a(G)$.
Case 2: Any biconnected subgraph of H has no more than k nodes. Case 2 follows from the following lemmata, whose proofs are non-trivial and omitted.

Lemma 8: $f \leq g(k + g - 1 - \alpha(H))$.

Suppose a node v is removed from H. If its removal separates H, then consider the component obtained by merging v with all the components except the largest. Call this component $C(v)$. If v's removal does not disconnect the graph, then let $C(v) = \{v\}$. If $|C(v)| \leq i$, then we say that v is *i-connected.*

Lemma 9: *Let H be a graph with β 1-connected nodes and no more than $2k$ nodes. If there are no biconnected components in H with more than $k + 1$ nodes, then there is a subgraph with $k + 1$ nodes and at least $\beta/4$ 1-connected nodes.*

Suppose that H has no more than $2k$ nodes and $k + g - \alpha(H)$ 1-connected nodes. Let H' be the subgraph found by lemma 9. H' has $k + 1 - \alpha(H')$ articulation nodes which, by the lemma, is at least $(k + g - \alpha(H))/4$. Since $a(G) \geq k + 1 - \alpha(H')$, $4a(G) \geq (k + g - \alpha(H))$. Combining the two lemmas, $4a(G) \geq (k + g - \alpha(H) \geq f/g$. □

Let G be a tree. Then, $a(G)$ becomes $\max_{T \in \mathcal{T}_{k+1}(G)} |\ell(T)| - 1$. For this important class of access graphs, we have:

Theorem 10: *When G is a tree, $c_{LRU,k}(G) = a(G) = c_k(G)$.*

Notice that LRU's servers will always be contiguous. 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. Below, we describe how this tokens are distributed, and prove that (i) at most $a(G)$ tokens are placed for any OPT fault, and (ii) All the LRU faults are serviced by servers with tokens.

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 10 follows from the following two lemmata.

Lemma 11: *At most $a(G)$ tokens are placed for any OPT fault.*

Lemma 12: *No LRU server without a token will service an LRU fault.*

Before proving the lemmata, we need a few facts.

Proposition 13: *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 up until 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 farthest in the future. This is always a leaf of *OPTtree*(t). Thus OPT's servers will be contiguous 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 14: *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 leaf 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. □

Proof of Lemma 11: 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. If the requested node is a leaf of the tree, then no token is placed for this leaf. Suppose that the requested node is an interior node of *LRUtree*(t). Since the requested node is a leaf of *OPTtree*(t), there is a path from the requested node to a leaf of *LRUtree*(t) that only contains nodes with lonely LRU servers. No token will be placed for this leaf because all the servers on this path have tokens. □

Proof of Lemma 12: Suppose some unmarked LRU server becomes the least recently used server at time t_2. The node where the server is located, v, is a leaf of $LRUtree(t_2)$. Let $T(v)$ be the subtree of G rooted at v that does not contain any other LRU server.

Proposition 15: *At time t_2, all of the lonely OPT servers are in $T(v)$.*

Proposition 15 implies Lemma 12 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 before the fault and the server on v will not be the least recently used. □

Proof of Proposition 15: Let $t_1(< t_2)$ be the last time node v was requested. No nodes in $T(v)$ are requested in the interval from t_1 to t_2, which we denote by (t_1, t_2). On the other hand all the nodes in $LRUtree(t_2)$ are requested in this interval.

Because all the lonely LRU servers have tokens, at time t_1, the number of OPT servers in $T(v) \geq$ the number of LRU servers in $T(v)$ without tokens.

No nodes in $T(v)$ are requested in the interval (t_1, t_2), so no node in $T(v)$ can be marked twice. Thus, the number of LRU servers without a token in $T(v)$ at time $t_1 \geq$ the number of tokens placed on nodes in $T(v)$ in the interval (t_1, t_2).

Consider $LRUtree(t)$ for some point in time in the interval (t_1, t_2). $LRUtree(t)$ has a leaf in $T(v)$. Whenever OPT incurs a fault in the interval (t_1, t_2), the first node without a token reached on the path from the 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 $(t_1, t_2) \geq$ the number of OPT faults in the interval (t_1, t_2).

We count the number of OPT faults by the number of nodes from which an OPT server leaves in the interval (t_1, t_2). Examine the set of nodes that are occupied by lonely LRU servers at time t_2. These nodes have been requested since time t_1 because each lonely LRU server has serviced a request since time t_1. For each one of these nodes, an OPT server has left that node in the interval (t_1, t_2). Furthermore, none of these nodes are in $T(v)$ because no node in $T(v)$ is requested in the interval (t_1, t_2). This yields that the number of OPT faults in the interval $(t_1, t_2) \geq$ the number of OPT servers that leave a node in $T(v)$ in the interval (t_1, t_2) plus the number of lonely LRU servers at time t_2.

Putting all the inequalities together: the number of OPT servers in $T(v)$ at time $t_1 \geq$ the number of OPT servers that leave $T(v)$ in the interval (t_1, t_2) plus the number of lonely LRU servers at time t_2. Thus the number of OPT servers remaining in $T(v)$ at time $t_2 \geq$ the number of lonely LRU servers at time t_2. So all the lonely OPT servers are in $T(v)$ at time t_2. □

Corollary 16: *When G is a path, $c_{LRU,k} = 1$.*

We omit the proof of the following theorem.

Theorem 17: *For any G with $n \geq k + 1$, $c_{FIFO,k} \geq (k + 1)/2$.*

Theorem 17 and the fact that LRU is k-competitive on any G [19] imply that $c_{LRU,k}(G) \leq 2c_{FIFO,k}(G)$ for all G. On many graphs $c_{LRU,k}(G)$ is much smaller. This provides an explanation of why LRU should be preferred to FIFO in practice.

LRU is used both as a paging strategy as well as a caching strategy. In caching, the number of blocks in the cache is 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 18: *For all G, (a) $c_{LRU,2}(G) = c_2(G)$; (b) $c_{LRU,4}(G) \leq (3/2)c_4(G)$.*

Proof of part (a) k=2: For every sequence σ and every on-line algorithm A with two servers, we show a valid sequence σ' such that for some $a \geq 0$, $A(\sigma') \geq LRU(\sigma) + a$ and $OPT(\sigma') \leq OPT(\sigma) + a$. We obtain

$$c_{A,2}(G) \geq \frac{A(\sigma')}{OPT(\sigma')} \geq \frac{LRU(\sigma) + a}{OPT(\sigma) + a} \geq \frac{LRU(\sigma)}{OPT(\sigma)}.$$

Simulate A and LRU on σ. Let z be the first node requested where A services a request differently from LRU. Let x and y be the two nodes requested before the request to node z. (x is requested before y). Since LRU always has its servers on the last two nodes requested, both A and LRU have servers on nodes x and y before the request to z. LRU uses the server on x to service the request to z and A uses the server on y. To construct σ', add to the sequence a request to node y and then z after the request to node z. Keep requesting y and then z until A's servers are alligned with LRU's. A incurs at least one extra fault. Then continue with σ until A's servers and LRU's servers are again unalligned. A's cost is at least the sum of LRU's cost and the number of places that requests are added to the sequence.

OPT's cost increases by at most the number of places that requests are added to the sequence. Suppose requests to nodes y and z are added to the sequence σ. There were requests to nodes y and z just before the place where the new requests are added. Suppose that in servicing σ OPT had intended to service the requests to nodes y and z with the same server, keeping its other server on some other node u. When servicing σ', OPT services the requests to y and z with different servers. OPT incurs at most one extra fault by not having a server on node u. □

Proof of part (b) k=4 (sketch): We consider four possibile cases:

1. The graph G has at least one vertex of degree four. Then, $c_4(G) \geq 3$ by Proposition 2. Further, for every graph G, $c_{LRU,4}(G) \leq 4$.

2. The graph G has a biconnected component of size five. Then, we can show that $c_4(G) \geq 3$. Further, for every graph G, $c_{LRU,4}(G) \leq 4$.

3. Not cases (1) and (2), and the graph G has at least one vertex of degree three. Then, $c_4(G) \geq 2$ by Proposition 2. It can be shown that in this case $c_{LRU,4}(G) \leq 3$.

4. Not cases (1), (2) and (3). In this case the graph G is either a path or an n-cycle for some $n > 5$. In both cases it can be shown that $c_{LRU,4}(G) \leq (4/3)c_4(G)$.

□

We conjecture that by a more refined analysis of Case (3) it can be proven that for all G, $c_{LRU,4}(G) \leq (4/3)c_4(G)$. This bound would be tight.

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 and chooses the unmarked server to use on a fault by vacating a blue node whose distance to the nearest red node (in G) is maximum. The intuition behind FAR is as follows: it is known [1, 15] 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. Let $M_G = \max_{T \in \mathcal{T}_k(G)} |\ell(T)|$, and let D_G be the maximum diameter of any connected k-node induced subgraph of G.

Theorem 19: *For any G and k, (a) $c_{FAR,k}(G) \leq 2M_G(1 + \lceil \log D_G \rceil)$. By Proposition 2 this is $\leq 2c_k(G)\lceil \log 2D_G \rceil \leq 2c_k(G)\lceil \log 2k \rceil$. (b) $c_{FAR,k}(G) \leq k$.*

Thus $c_{FAR,k}(G)$ is within $2\lceil \log 2k \rceil$ of $c_k(G)$ on every G, a performance LRU and FIFO cannot match. Note that (a) 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. By (b), FAR is optimal in the Sleator-Tarjan model (G is the complete graph), including the constant. For nodes u, v and a set of nodes S, let $d(u, S)$ denote the length of the shortest path from u to any member of S, and let $d(u, v) = d(u, \{v\})$.

Proof of Theorem 19: (a) Let g be the number of new nodes in the current phase. By Proposition 4, 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. We divide the current phase into roughly $\log D_G$ *sub-phases*, showing that at most gM_G blue nodes are vacated during each sub-phase.

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 marked node at the time it is vacated. The sequence $d_1, d_2, \ldots$ is non-increasing. Let i_2 be the smallest index such that $d_{i_2} \leq d_1/2$; 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 $j \geq 2$, begins at the smallest index i_j such that $d_{i_j} \leq d_{i_{j-1}}/2$. Let S_j be the set of nodes that are marked 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 white 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 M_G, and this will imply a bound of gM_G on the number of nodes vacated in the sub-phase. For any two reps u, v, $d(u, S_j) > d_{i_j}/2$, $d(v, S_j) > d_{i_j}/2$ and $d(u, v) > d_{i_j}/2$. (The last follows from the fact that the second of $\{u, v\}$ to be vacated must be at distance $> d_{i_j}/2$ from the first, which is in the set of marked nodes by then.) Imagine contracting S_j to a node; we now argue that a tree can be grown from S_j with each rep as a leaf of this tree. Assume not: that is any path to rep v must pass through some other rep. Let the shortest path from S_j to v pass through rep u. Then $d(v, S_j) = d(v, u) + d(u, S_j) >$

d_{i_j}, implying that v should have been vacated in a prior sub-phase. The number of nodes in S_j together with the set of reps is $\leq k$, so that the number of leaves of this tree (i.e., the number of reps) is $\leq M_G$.

The proof of (b) follows from Proposition 4. □

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 the ith sub-phase.

There exists a tree T such that $r_1 = O(|\ell(T)|)$. This follows from the following graph-theoretic proposition, whose proof is omitted.

Proposition 20: *Let $n_2(G)$ be the number of nodes of degree $\neq 2$ in a graph G. Then, there exists a tree $T \subseteq G$ such that $|\ell(T)| = \Omega(n_2(G))$.*

Proposition 21: *Let $\mathcal{P}$ be the set of paths consisting of the unmarked blue nodes at the end of the first sub-phase. Then*

$$r_2 = O(\sum_{P \in \mathcal{P}} v(P)),$$

where $v(P)$ is $1 + \lfloor \log |P| \rfloor$.

Proof: Each request is served by the server at the mid-point of some path $P \in \mathcal{P}$. 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 20 as a tree in a vine decomposition of G, there exists a vine decomposition $(T, \mathcal{P})$ for which $\sum_{i \geq 1} v(P_i) + n_T \geq r_1$. Taking another vine decomposition whose set of paths $\mathcal{P}$ is the set of blue paths at the end of the first sub-phase, we have a vine decomposition $(T, \mathcal{P})$ for which $\sum_{i \geq 1} v(P_i) + n_T \geq r_2$. Since the lower bound is given by the maximum over all vine decompositions $c_k(G)$ is $\Omega(r_1 + r_2)$.

Theorem 22: *For any G and $k = n-1$, (a) $c_{TS,k}(G)$ is $O(c_k(G))$, and (b) $c_{TS,k}(G) \leq k$.*

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 dag 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 directed cycle C: we identify v and one node in 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.

Notice that a GOTO statement could not be captured by such a graph. We now present a variant of FAR which we call 2FAR, and prove that $c_{2FAR,k}(G)$ 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, it 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 strongly connected component containing v and is 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 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 at the node that is the predecessor of u in C_i (which is in fact furthest from the marked nodes) is used.

Theorem 23: *For any structured program graph G in which every strongly connected component has size $\leq k+1$, $c_{2FAR,k}(G) = c_k(G)$.*

We omit the proof here, and can show that a natural extension of 2FAR can be shown to have competitiveness at most $gc_k(G)$ whenever every strongly connected component of G has size at most $k+g$. We suspect that the correct bound is much closer to $c_k(G)$. Note that LRU cannot achieve this performance (because of the $k+1$-node cycle).

5.2. Randomized Paging Algorithms

We return to undirected access graphs. It has been shown [6, 15] that in the Sleator-Tarjan model, the competitveness of the optimal randomized on-line paging algorithm is H_k (the kth harmonic number) against 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 [6] is within a constant factor of the optimal randomized algorithm for the important case when G is a tree.

Proposition 24: *No randomized paging algorithm achieves a competitiveness less than $\lceil \log(k+1) \rceil /2$ on the $k+1$-node circle against an oblivious adversary.*

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

Theorem 25: *For any randomized algorithm R, $c_{R,k}(G) \geq H_{M-1}$.*

Consider the following randomized algorithm for tree access graphs, which is a variant of that in [6]. Look at the tree $t \in T_k(G)$ induced by the nodes marked in the previous phase. On a miss, vacate the server at a random unmarked leaf in this tree. It is easy to show that this achieves an upper bound within a factor of two of the lower bound of Theorem 25.

5.3. Open Problems

By Theorem 17, LRU is never more than twice as bad as FIFO, and is often better. It would be worth removing the factor of two:

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

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

Open Question 2: *How close to optimal is $c_{FAR,k}(G)$?*
There is a graph G for which $c_{FAR,k}(G) \geq (4/3)$ $c_k(G)$, so it is not optimal on all graphs.

Open Question 3: *For directed graphs, how close to $c_k(G)$ is $c_{2FAR,k}(G)$? Is there a near-optimal algorithm for 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 3):

Open Question 4: *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 against an oblivious adversary?*

Acknowledgements

We thank Don Coppersmith, Amos Fiat and Howard Karloff for their comments and suggestions.

References

[1] L.A. Belady. A study of replacement algorithms for virtual storage computers. *IBM Systems Journal*, 5:78–101, 1966.

[2] S. Ben-David, A. Borodin, R.M. Karp, G. Tárdos, and A. Wigderson. On the power of randomization in on-line algorithms. In *Proc. 22nd Annual ACM Symposium on Theory of Computing*, pages 379–388, 1990.

[3] A. Borodin, N. Linial, and M. Saks. An optimal online algorithm for metrical task systems. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 373–382, 1987.

[4] P.J. Denning. Working sets past and present. *IEEE Trans. Software Engg.*, SE-6:64–84, 1980.

[5] D. Ferrari. The improvement of program behavior. *IEEE Computer*, 9:39–47, November 1976.

[6] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. On competitive algorithms for paging problems. To appear in *Journal of Algorithms*, 1991.

[7] P.A. Franaszek and T.J. Wagner. Some distribution-free aspects of paging performance. *Journal of the ACM*, 21:31–39, 1974.

[8] D. Hatfield and J. Gerald. Program restructuring for virtual memory. *IBM J. Syst. and Tech.*, 10:168–192, 1971.

[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*, pages 508–511, 1989.

[10] A. R. Karlin, M. S. Manasse, L. Rudolph, and D.D. Sleator. Competitive snoopy caching. *Algorithmica*, 3(1):70–119, 1988.

[11] T. Kilburn, D.B.G. Edwards, M.J. Lanigan, and F.H. Sumner. One-level storage system. *IRE Trans. Elect. Computers*, 37:223–235, 1962.

[12] A. Lempel, S. Even, and I. Cederbaum. An algorithm for planarity testing of graphs. In *Proc. Int. Symp. on Theory of Graphs; P. Rosenstiehl Ed.*, pages 215–232. Gordon and Breach, 1967.

[13] P.A.W. Lewis and G.S. Shedler. Empirically derived models for sequences of page exceptions. *IBM J. Res. and Develop.*, 17:86–100, 1973.

[14] M.S. Manasse, L.A. McGeoch, and D.D. Sleator. Competitive algorithms for on-line problems. *Journal of Algorithms*, 11:208–230, 1990.

[15] L.A. McGeoch and D.D. Sleator. A strongly competitive randomized paging algorithm. Technical Report CMU–CS–89–122, Carnegie-Mellon University, Pittsburgh, PA, 1989. Submitted to *Algorithmica.*

[16] L.A. McGeoch, D.D. Sleator, and C. Tomasi. Decision procedures for competitive algorithms. In preparation.

[17] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms. In *16th International Colloquium on Automata, Languages, and Programming*, volume 372 of *Lecture Notes in Computer Science*, pages 687–703. Springer-Verlag, July 1989. Revised version available as an IBM Research Report.

[18] G.S. Shedler and C. Tung. Locality in page reference strings. *SIAM Journal on Computing*, 1:218–241, 1972.

[19] D.D. Sleator and R.E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, February 1985.

[20] J.R. Spirn. *Program Behavior: Models and Measurements.* Elsevier Computer Science Library. Elsevier, Amsterdam, 1977.

[21] N. Young. Competitive paging as cache-size varies. In *Proc. Second ACM-SIAM Symposium on Discrete Algorithms*, pages 241–250, 1991.