Strongly Competitive Algorithms for Paging with Locality of Reference

Sandy Irani* Anna R. Karlin † Steven Phillips ‡

July 5, 1995

Abstract

What is the best paging algorithm if one has partial information about the possible sequences of page requests? We give a partial answer to this question, by presenting the analysis of strongly competitive paging algorithms in the access graph model. This model restricts page requests so that they conform to a notion of locality of reference, given by an arbitrary access graph.

We first consider optimal algorithms for undirected access graphs. Borodin et al. [2] define an algorithm, called FAR, and prove that it is within a logarithmic factor of the optimal on-line algorithm. We prove that FAR is in fact strongly competitive, i.e. within a constant factor of the optimum. For directed access graphs, we present an algorithm that is strongly competitive on *structured program graphs* — graphs which model a subset of the request sequences of structured programs.

Keywords: Analysis of Algorithms, On-line Algorithms, Competitive Analysis, Paging, Locality of Reference.

1 Introduction

Many computer systems have a two-level store of memory consisting of a small fast memory and a large slow memory. The abstraction of a large, fast virtual memory is often implemented by dividing the slow memory into pages, and keeping those pages that are likely to be referenced soon in the fast memory. A *page fault* occurs when a reference is made to a page that is not resident in fast memory. Handling a page fault is typically expensive since the page must be brought into fast memory. A *page replacement strategy* specifies which page to replace on a page fault.

If the future sequence of page requests is known in advance, the optimal page replacement strategy is clear: replace the page whose next request is farthest in the future [1]. Unfortunately, the decision about which page to replace must usually be made *on-line*, without detailed information about future requests.

How can we analyze such an on-line algorithm? Straightforward worst-case analysis is useless: if arbitrary request sequences are allowed, then an adversary that always requests the last discarded page can force any paging algorithm to fault on each request. Average-case analysis is also problematic, since it requires a statistical model of the sequence of requests. It is extremely difficult to devise a realistic model, since the pattern of accesses changes dynamically with time and

*Dept of Computer Science, UC Berkeley, Berkeley, CA.

†DEC Systems Research Center, 130 Lytton Ave., Palo Alto, CA 94301.

‡Dept of Computer Science, Stanford University and DEC Systems Research Center. Partially supported by NSF Grant CCR-9010517 and an OTL grant.

with different applications. Nonetheless, several of the early analyses of paging algorithms were performed assuming a fixed probability distribution on the request sequences [5, 11, 1].

In order to get around these problems, the notion of *competitive analysis* was introduced by Sleator and Tarjan [10]. An on-line page replacement strategy A is said to be c-*competitive* if there exists a constant β such that for every request sequence σ,

$$A(\sigma) \leq c \cdot \mathrm{OPT}(\sigma) + \beta,$$

where $A(\sigma)$ is the cost (i.e. the number of faults) incurred by the algorithm A in processing the request sequence σ and $\mathrm{OPT}(\sigma)$ is the cost incurred by the optimal off-line algorithm in processing σ. The *competitiveness* of A, denoted by c_A, is the infimum of c such that A is c-competitive.

Competitive analysis avoids the assumptions of probabilistic analysis, and has the power of differentiating between different paging algorithms. Sleator and Tarjan showed that no deterministic paging algorithm can achieve a competitiveness better than k, where k is the number of pages of fast memory. They also showed that some practical algorithms, such as first-in-first-out (FIFO) and least-recently-used (LRU) are k-competitive, and hence best possible in their model.

1.1 Locality of Reference.

The Sleator-Tarjan results conflict with practical experience on paging in at least two ways. First, FIFO and LRU have the same competitiveness, even though in practice LRU usually outperforms FIFO. Secondly, LRU usually incurs much less than k times the optimal number of faults, even though its competitiveness is k.

The reason for the practical success of LRU has long been known: most programs exhibit *locality of reference* [1, 3, 9]. This means that if a page is referenced, it is more likely to be referenced in the near future (temporal locality) and pages near it in memory are more likely to be referenced in the near future (spatial locality). Indeed, a two-level store is only useful if request sequences are *not* arbitrary.

Motivated by these observations, Borodin, Irani, Raghavan and Schieber [2] proposed a technique for incorporating locality of reference into the traditional Sleator-Tarjan framework. Their notion of an *access graph* limits the set of request sequences the adversary is allowed to make.

An access graph $G = (V, E)$ for a program is a graph that has a vertex for each page that the program can reference. Locality of reference is imposed by the edge relation – the pages that can be referenced after a page p are just the neighbors of p in G or p itself. Thus a request sequence σ must be a walk on G. The specific walk that is generated is determined by the data given to the program. The definition of competitiveness remains the same as before, except for this restriction on the request sequences. Let $c_{A,k}(G)$ denote the competitiveness of an on-line algorithm A with k pages of fast memory on the access graph G. We denote by $c_k(G)$ the infimum (over on-line algorithms A with k pages of fast memory) of $c_{A,k}(G)$. Thus $c_k(G)$ is the best that any on-line algorithm can do. We say an algorithm A is *strongly competitive* if $c_{A,k}(G) = O(c_k(G))$. [1]

An access graph may be either directed or undirected. An undirected access graph might be a suitable model when the page reference patterns are governed by the data structures used by the program. For example, if a program performs operations on a tree data structure, and the mapping of the tree nodes to pages of virtual memory represents a contraction of a tree, then the appropriate access graph might be a tree. Alternatively, if we were to completely ignore data, and focus only on the flow of control inherent in the structure of the program, a directed access graph might be a more suitable model.

[1]This terminology differs slightly from that of [6, 8], where strongly competitive was defined as achieving the competitive ratio $c_k(G)$.

Unfortunately, modeling access patterns is not as simple as these examples would suggest. See Section 4 for a discussion of the limitations of the model.

1.2 Summary of Results.

In this paper, we show that there are strongly competitive page replacement strategies in two important settings: 1) undirected access graphs and 2) directed access graphs representing the stream of instruction references made by a structured program.

Undirected Access Graphs:

Borodin *et al.* proved that on any undirected access graph G, $c_{\mathrm{LRU},k}(G) \leq 2c_{\mathrm{FIFO},k}(G)$, and that LRU is often better than FIFO. However, on some graphs the competitiveness of both LRU and FIFO is much greater than $c_k(G)$. For example, it has been observed [7] that LRU and FIFO behave badly on loops just larger than k. This is substantiated by the following result in the access graph model: if G is the $k+1$-node cycle, then $c_k(G) = \lceil \log(k+1) \rceil$, while $c_{\mathrm{LRU},k} = k$. Thus LRU can be far from optimal among on-line algorithms.

We seek a *universal* algorithm, with a succinct description, whose competitiveness is close to $c_k(G)$ on *every* graph G. Importance is lent to this question by recent operating system research, in which facilities are provided for user-defined page replacement strategies [12, 13]. When these facilities are available, one must consider the following question: What is the best paging algorithm if one has partial information about the possible sequences of page requests? When the partial information is represented by an access graph this question asks for a universal paging algorithm that is *strongly competitive.*

Borodin *et al.* describe a simple and natural algorithm called FAR, and prove that it achieves a competitiveness within $O(\log k)$ of $c_k(G)$for every graph G. They leave open the question of finding an algorithm with competitive ratio $O(c_k(G))$. Our main result for undirected access graphs is that FAR is optimal, within a constant factor, among on-line algorithms for paging with locality of reference.

Theorem 1 *For any graph G and memory size k, the algorithm FAR is strongly competitive, i.e.*

$$c_{FAR,k}(G) = O(c_k(G)).$$

The proof relies on a graph decomposition called a *vine decomposition* [2]. Borodin *et al.* show how to find a lower bound on $c_k(G)$ using a vine decomposition of G. We show that this lower bound is essentially optimal, by using the fault pattern of FAR to construct a vine decomposition of G.

Directed Access Graphs:

In Section 3 we consider optimal algorithms for directed access graphs that represent a subset of the stream of instruction references made by a structured program. We use the notion of *structured program graphs* as in [2].

Structured program graphs (spg's) are defined recursively as follows: Every spg has a designated *start* node and a designated *stop* node. A single directed edge is an spg, where the node with out-degree one is the start node and the node with in-degree one is the stop node. More complex spg's can be constructed by applying the following rules.

- Two spg's G_1 and G_2 can be combined to get a new spg, by identifying the start node of G_1 and the stop node of G_2 (serial composition).
- Two spg's G_1 and G_2 can be combined to get a new spg, by identifying identifying the stop nodes of G_1 and G_2. (branching statement).
- A node v in an spg can be identified with a node u of a directed cycle to yield a new spg (loop).

Notice that there is no way to represent arbitrary 'GOTO' statements. A significant limitation of this model is that the definition does not allow branching within loops.

Borodin *et al.* analyze a simple generalization of FAR, called 2FAR, that is optimal for spgs in which all strongly connected components have at most $k+1$ nodes. We introduce a variant of 2FAR, called EVEN, and prove that it is strongly competitive for all spg's.

Theorem 2 *The algorithm* EVEN *is strongly competitive on the class of structured program graphs. In other words, for any structured program graph G, $c_{k,\mathrm{EVEN}}(G) = O(c_k(G))$.*

2 Undirected Access Graphs

2.1 Background.

We review some terminology and results on competitive paging. A sequence of page requests and the resulting behavior of a paging algorithm can be divided into *phases*. At the beginning of a phase, all pages are unmarked. When a page is *requested* (or *hit*) it is said to be *marked.* A phase ends just before the request to the $k+1^{st}$ distinct node. At the end of a phase all nodes become unmarked again. We call the nodes requested in this phase but not in the previous phase *new* nodes. A node is said to be *evacuated* if it has been evicted from the fast memory during the phase. If a page is in the fast memory, we will say it is covered by a *server*.

The algorithm FAR is a *marking algorithm.* A marking algorithm is an algorithm that, on a fault, always evicts an *unmarked* page. The unmarked page that FAR chooses to evict is the page that is *furthest* (in the access graph) from the set of marked nodes.

We think of the sequence of page requests as being generated by an *adversary*, and say that the *cost* of the adversary is the cost of the offline algorithm on the request sequence.

Proposition 3 *(a) If g_i new nodes are requested in the i'th phase, then the cost of the adversary during the first i phases is at least $(\sum_{j=1}^{i} g_j)/2$, and at most $\sum_{j=1}^{i} g_j$. (b) If A is a marking algorithm then for any graph G, $c_{A,k}(G) \le k$.*

The proof of Part (a) is due to Fiat *et al.* [4], while Part (b) is due to Karlin *et al.* [6].

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 pairwise edge-disjoint simple paths in H (without loops), called *vines* such that (i) the nodes of H are partitioned between T and the interior nodes of the P_i's; (ii) each endpoint of each path is a node in T. If a tree node is the endpoint of a vine, it is called an *anchor*. Let n_T be the number of leaves of T that are not anchors. The *value* of a path P, denoted $v(P)$, is defined to be $\lceil \log |P| \rceil$, where $|P|$ is the number of edges in P. The *value* of the vine decomposition $\mathcal{V}(H)$, denoted $v(\mathcal{V}(H))$ (or $v(\mathcal{V})$ for short) is

$$v(\mathcal{V}(H)) = \sum_i v(P_i) + n_T - 1.$$

Lastly, if $\mathcal{C}$ is a cycle on $k+g$ nodes, $1 \le g \le k$, then the *value* of $\mathcal{C}$ is $v(\mathcal{C}) = \lfloor \log k - \log g \rfloor / 2$.

Theorem 4 *1. For any graph G and memory size k, let $\mathcal{V}$ be a vine decomposition on $k+1$ nodes in G. Then $c_k(G) \geq v(\mathcal{V})$.*

2. Let G be a graph containing a cycle $\mathcal{C}$ on $k+g$ nodes, $1 \leq g \leq k$. Then $c_k(G) \geq v(\mathcal{C})$.

Proof: The lower bounds are proved by describing an adversary who walks on the access graph, causing the online algorithm to fault a large number of times before the phase ends. This proves something a little stronger, that in *each phase*, the ratio of faults made by the online and offline algorithms is at least the stated lower bound.

The proof of Part 1 is due to Borodin *et al.* [2]. For completeness we give an outline of the proof here. W.l.o.g. assume that G has $k+1$ nodes, so that the adversary incurs at most 1 fault per phase. An on-line algorithm A always leaves one node of G uncovered – this node is called the *hole.* The bound on $c_k(G)$ follows from noting that during a phase, the adversary can make A incur $v(\mathcal{V})$ faults. When A moves its hole to a new position following a fault, the adversary (and the request sequence) walks to that position, causing A to fault there. If the new location of the hole is a tree node, the adversary walks to the hole on a path that contains only marked nodes or internal tree nodes. If the new location of the hole is an interior node of a vine v, the adversary walks to the hole on a path that contains only marked nodes, internal tree nodes, anchors, and at most half the unmarked nodes on the vine v. The hole can be reached at least n_T times by requests to non-anchor leaves, and at least $v(P) = \lceil \log |P| \rceil$ times by requests to each vine P.

For Part 2, assume w.l.o.g. that G is just a cycle $\mathcal{C}$ on $k+g$ nodes. There are g "holes" (nodes where the online algorithm A doesn't have a server) in G at any time. A phase starts with the adversary and A covering the same set of contiguous vertices. If A ever has a hole on a marked node, the adversary requests the path to the hole, passing only through marked nodes. The phase begins with the adversary issuing requests to all g of the uncovered nodes. At any point, the set of unmarked nodes forms a contiguous path. The adversary then repeats the following pattern: he requests all the nodes in whichever half of the path of unmarked nodes has more holes, causing the on-line algorithm to incur at least $g/2$ faults. When the pattern has been repeated $i = \lfloor \log k - \log g \rfloor$ times, the path of unmarked nodes has length $\lfloor k/2^i \rfloor < 2g$, and the adversary simply requests unmarked nodes till g remain, and the phase ends.

The adversary incurs only g faults during this process: at the start of the phase he vacates the g nodes that will remain unmarked at the end of the subsequence. Therefore the ratio between the number of faults incurred by the on-line algorithm and the adversary is $i/2 = \lfloor \log k - \log g \rfloor /2$. Since the situation at the end of this phase is the same as at the start, the adversary can repeat this behavior ad infinitum, hence $c_k(G) \geq \lfloor \log k - \log g \rfloor /2 = v(\mathcal{C})$. □

2.2 Proof of FAR's Strong Competitiveness.

In this section we prove that FAR is a strongly competitive algorithm:

Theorem 5 *For any undirected access graph G, $c_{FAR,k}(G) = O(c_k(G))$.*

For the entire proof, we restrict our attention to a single phase of the request sequence. Suppose that g new nodes are requested in the phase. By Proposition 3, the adversary incurs at least $g/2$ faults for this phase, amortized. The proof works by showing that there is a vine decomposition or large cycle whose value is at least c/g times the number of faults incurred by FAR in the phase, for some absolute constant c. The lower bound of Theorem 4 then implies that the number of faults incurred by FAR during the phase is $O(g \cdot c_k(G))$. Because this is true for all phases, $c_{FAR,k}(G) = O(c_k(G))$.

We use the concept from [2] of *representatives* or *reps* for short. Each time FAR incurs a fault, it must vacate some node. Consider the sequence of nodes that are vacated during the phase: we divide this sequence into blocks of $g+1$ nodes that are vacated successively. The number of new nodes requested in the phase is g, so at any time during the phase, the number of nodes that have been evicted so far during the phase and are currently outside the fast memory is at most g. Therefore there is at least one node of each block that is marked before *any* node of the subsequent block is vacated by FAR. We pick such a node for each block and call it a *rep*.

Suppose that the number of reps for this phase is R. We need to construct a vine decomposition on $k+1$ nodes or a cycle whose value is $\Omega(R)$.

Preliminaries

Let G' be the directed subgraph of G containing every node that is marked during the phase, with a directed edge from u to v if the first request at v immediately follows a request at u. The graph G' is a directed tree, rooted at the first node requested during the phase, and a directed path from x to y in G' implies that x was marked before y.

The following definitions are fundamental to the rest of the proof.

Definition 1 *(i) A node in G' is a* chain node *if it has degree 2 in G and both indegree and outdegree 1 in G'. Nodes that are not chain nodes are called* non-chain nodes.
(ii) A chain is a directed path in G' whose interior nodes are chain nodes, and whose endpoints are not chain nodes.

There is a natural reason for not allowing a node whose degree is more than 2 in G to be a chain node in G'. The reason is that otherwise a chain might not be "self-contained", in the following sense: a chain node that had degree at least 3 in G could be close to a marked node outside the chain, but far from any marked node inside the chain. Excluding this case makes it possible to analyze chains in isolation.

We divide the reps into two types.

Definition 2 *A rep is of type 1 if it is the last or one of the first five reps on a chain, or it is a non-chain node. Any other rep is of type 2 (i.e. any rep in the middle of a chain). Let R_1 and R_2 be the number of type 1 and type 2 reps respectively.*

We have two constructions: the first is a vine decomposition which is a tree on $k+1$ nodes with $\Omega(R_1)$ leaves. The other construction is a vine decomposition $\mathcal{V}$ on $3k/2$ nodes whose vines have total value $\Omega(R_2)$. We use $\mathcal{V}$ to find a vine decomposition on $k+1$ nodes whose vines have total value $\Omega(R_2)$ or to find a large cycle whose value is $\Omega(R_2)$. Since one of R_1 and R_2 has value $\Omega(R)$, the two constructions will complete the proof of Theorem 5.

Constructing a Vine Decomposition with Value $\Omega(R_1)$.

Theorem 6 *The graph G contains a subgraph on $k+1$ nodes which is a tree with $\Omega(R_1)$ leaves.*

Proof: Charge each type 1 rep that is on a chain to the far endpoint of the chain, and charge reps that aren't on chains to themselves. Each non-chain node is charged at most 6 times (because G' is a directed tree, so each node is the far endpoint of at most one chain). Hence the number of non-chain nodes is at least $R_1/6$. Now a non-chain node is either a leaf of G', the root of G', or has degree 3 or more in G. Let UG be the undirected version of the graph G'. Root the tree UG at the root of G'.

Case 1: At least half of the non-chain nodes in G' either:

1. are leaves or the root in UG
2. or have degree 3 or more in UG
3. or have degree 2 in UG and a parent of degree 3 or more in UG.

If v satisfies condition 3, then its child in UG doesn't (and it has a child since it is not a leaf). And the first descendent of v that does not have degree 2 satisfies condition 1 or 2. Therefore if α nodes satisfy condition 3, then at least α nodes in the tree satisfy conditions 1 or 2. And so at least a quarter of the nonchain nodes in the tree satisfy conditions 1 or 2. Thus, at least $R_1/24 - 1$ nodes are leaves in UG or have degree 3 or more in UG.

The average degree of nodes in a tree is less than 2, since any tree with n nodes has $n-1$ edges. Therefore, at least half of the nodes not of degree 2 in UG are leaves, and UG is the desired tree.

Case 2: Otherwise, at least half of the non-chain nodes in G' have degree 3 or more in G, but degree 2 in UG and a parent of degree 2 in UG: Let S be a maximal set of vertices in UG that have degree three or more in G, degree two in UG, and whose parent in G' has degree 2 in UG and is not in S. It is easy to see that $|S| = \Omega(R_1)$. For each $v \in S$, let $a(v)$ be an arbitrary vertex that is a neighbor of u in G but not in UG.

For each $w \in G \setminus UG$, let $num(w) = |\{v|a(v) = w\}|$. For each such w, if $num(w)$ is 1 or 2, then add the node w to UG together with a single edge to one of the nodes $v \in UG$ such that $a(v) = w$. Thus w becomes a leaf of UG. Alternatively if $num(w) \geq 3$, then add the node w to UG together with edges to all the nodes $v \in UG$ such that $a(v) = w$. For each of these nodes v, except for the one closest to the root of UG (breaking ties arbitrarily), sever the edge from v to its parent $p(v)$ in G'. Since $p(v)$ had degree 2 in UG, it becomes a leaf of UG, and UG remains a tree.

Since UG contains at most one new node for each node in S, at most $k/2$ nodes can be added to UG by this process. Finally, since at least $|S|/2$ leaves were added to UG, and $|S| = \Omega(R_1)$, UG has $\Omega(R_1)$ leaves.

We can then prune the tree until there are $k+1$ nodes, while keeping the number of leaves $\Omega(R_1)$. This can be done by first finding a node v whose removal from T results in components of size at most k. To find v, pick an arbitrary node u. Consider the tree rooted at u. If one of the children of u is the root of a subtree with more than k nodes, reset u to be that child. Keep resetting u until all the subtrees rooted at children of u have at most k nodes. Since the graph has at most $2k$ vertices, every time u is reset, the maximum number of nodes in a subtree rooted at a child of u decreases. Pick v to be the final u.

Gather the subtrees of v into groups as follows: consider each subtree and put it in the smallest numbered group possible so that no group has more than k nodes. There at most four groups since the sum of the nodes in any two groups is more than k. One group must contain $\Omega(R_1)$ leaves of T. This group can be easily extended to a connected subgraph of T containing $k+1$ nodes, giving the desired tree.

□

Constructing a Vine Decomposition with Value $\Omega(R_2)$.

The first step towards finding a vine decomposition or cycle whose value is $\Omega(R_2)$ is to show that the separation between type 2 reps decreases exponentially along directed paths in G'.

Definition 3 *For a chain node v, let $n(v)$ be the distance from v to the end of its chain.*

Lemma 7 *(i) If v, w and z are three consecutive reps on a chain, then $n(z) \leq n(v)/2$.*

(ii) If y is the second last rep on a chain in P and c is the third rep on a later chain in P, then $n(c) \leq n(y)$.

(iii) If P is a directed path in G', containing m type 2 reps, then $m \leq 2\lfloor \log l_1 \rfloor$, where l_1 is the length of the first chain in P containing a type 2 rep.

Proof:

Part (i) is proven as follows: Since v and w are consecutive reps on a chain, v is marked before w is evicted. Therefore, if w' is the last node on w's chain to be marked before w is evicted, then w' lies between v and w, and the unmarked portion of the chain extends from w' until the end of the chain at the time w is evicted. Note, however, that some of these nodes may not have servers on them. Since FAR evicts an unmarked node furthest from the set of marked nodes, if w is less then half the way from w' to the end of the chain, then every vertex between w and the midpoint must be a hole. Furthermore, none of them can be reps since any subsequent rep on the chain must be vacated after w is marked. Therefore, z must be beyond the midpoint of the chain from w' to the end and so $n(z) \leq n(w')/2 \leq n(v)/2$.

For Part (ii), suppose that z is the last rep on y's chain, and that a and b are the first two reps on c's chain (so the sequence of reps on chains in P is $\ldots, y, z, a, b, c, \ldots$). Since y is marked before z is evicted, z is at distance at most $n(y)$ from the set of marked nodes at the time it is evicted. Also, by Part (i), $n(c) \leq n(a) - n(c)$. Now suppose that $n(c) > n(y)$. Then $n(a) - n(c) > n(y)$ also. But then the server at c is further from the set of marked nodes at the time z is evicted than z is, and would be vacated before z, so c cannot be a rep, a contradiction. So $n(c) \leq n(y)$.

For Part (iii), let v_i, $1 \leq i \leq m$ be the sequence of type 2 reps on P. Let v_0 is the fifth type 1 rep on the first chain of P containing type 2 reps. By Parts (i) and (ii) we have $n(v_{i+1}) \leq n(v_i)/2$ or $n(v_{i+2}) \leq n(v_i)/2$, for $0 \leq i \leq m-2$. Hence $n(v_m) \leq n(v_0)/2^{\lfloor \frac{m}{2} \rfloor}$. Since there are four type 1 reps on the first chain before v_0, we have by Part (i) that $n(v_0) \leq l_1/4$. Therefore, $n(v_m) \leq l_1/(2^{\lfloor \frac{m}{2} \rfloor + 2})$. Lastly, $n(v_m) \geq 1$, so $1 \leq l_1/(2^{\lfloor \frac{m}{2} \rfloor} + 2)$. Rearranging, we obtain $m \leq 2\lfloor \log l_1 \rfloor$.

□

Because of this lemma, we know that the number of type 2 reps on any chain is logarithmic in the length of the chain. Therefore if we could make all chains be vines in some vine decomposition, then the vine decomposition would have value $\Omega(R_2)$, and we'd be done. The problem is that in order to connect the vines, we may need to use some chains to construct the tree, leaving only a subset of the chains to become vines. The following procedure constructs a vine decomposition $\mathcal{V}$ (possibly containing a small number of additional nodes from outside G') for which the value of the chains that become vines is $\Omega(R_2)$.

The procedure keeps a contraction $\tilde{G}$ of G', where each edge of $\tilde{G}$ is part of a chain of G' that contains some type 2 reps. [2] It never contracts only part of a chain, so we can refer to a chain of G' that hasn't been contracted yet as a "chain of $\tilde{G}$" (though its endpoints in $\tilde{G}$ may be supernodes). The contraction $\tilde{G}$ is always a tree directed from the root, as contracting preserves this property.

The procedure repeatedly identifies a chain $\mathcal{C}$ of $\tilde{G}$ that will be a vine $\mathcal{V}$, and describes the path between the endpoints of $\mathcal{C}$ in the tree of $\mathcal{V}$. This path may contain some chains of $\tilde{G}$ (and hence G'), but we use Lemma 7 to show that $v(\mathcal{C})$ is at least a constant fraction of the number of type 2 reps on the path. The identified chain and the chains used in the path are then contracted, and the process continues.

[2] To *contract* an edge (u, v) in a graph means that u and v are merged into a *supernode*. Edges to and from u and v are replaced by edges to and from the supernode, with duplicates deleted. A contraction of a graph is obtained by repeatedly contracting edges.

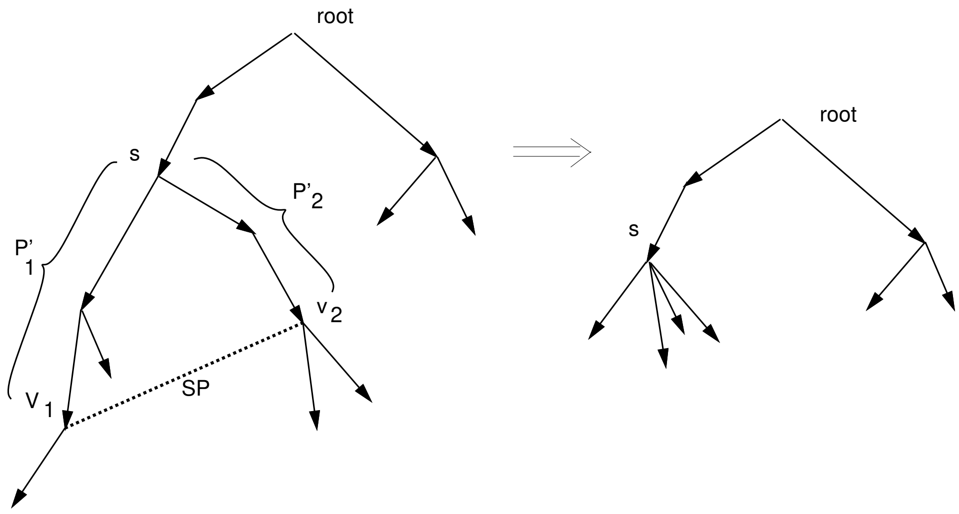

Figure 1: An iteration of the loop of FindVineDecomp

Procedure *FindVineDecomp*

Input: Undirected graph G, directed graph G' on nodes marked during the phase, and a record of FAR's faults during the phase.

Output: A vine decomposition $\mathcal{V}$ in G.

Initialization: First set $\tilde{G} = G'$. Then contract edges not in chains of G', and contract any chains that don't contain any type 2 reps. All edges so contracted are edges of the tree of $\mathcal{V}$. $\tilde{G}$ now consists of chains separated by (super)nodes.

Loop: Call the (super)node containing the first node marked in the phase *Root*. Repeat the following steps until *Root* is the only node in $\tilde{G}$.

1. Let v_1 and v_2 be two different (super)nodes in $\tilde{G}$. Let SP be the shortest path in G from any node contained in v_1 to any node contained in v_2, that doesn't use any edges of G' (if one exists). Choose v_1 and v_2 such that SP exists and $|SP|$ is minimal.

   Let P_1 and P_2 be the directed paths in $\tilde{G}$ from *Root* to v_1 and v_2 respectively. Let s be the last supernode contained in both P_1 and P_2. Let P_1' be the portion of P_1 extending from s on, and similarly for P_2'. (See Figure 1. Edges in the picture correspond to chains in $\tilde{G}$.)

2. Now assume w.l.o.g. that P_1 has more type two reps than P_2. Let $\mathcal{C}$ be the first chain of P_1: $\mathcal{C}$ is identified as a vine of $\mathcal{V}$, and the edges in SP, P_1 (except for $\mathcal{C}$) and P_2 are identified as edges in the tree of $\mathcal{V}$. In $\tilde{G}$ the paths P_1 and P_2 contracted (see Figure 1).

End *FindVineDecomp*.

There must always be a pair of supernodes v_0 and v_1 for which SP exists, for the following reason. Let v be a leaf in $\tilde{G}$, and let C be the chain leading to v. If all paths from v to nodes marked earlier than those on C run through C, then there could be no more than one rep in C and v, by the definition of FAR — a contradiction.

The following lemma bounds the length of SP.

Lemma 8 *In any iteration of the loop, the length of SP is at most $|\mathcal{C}|/2$.*

Proof: The proof is similar to the proof of Lemma 7. Let x, y and z be the last three reps (type 1 or 2) on the last chain of P_1. At the time y is evicted, x is marked, so at that time there must be some path Q of length at most $n(x) - n(y)$ from z, along the chain away from y, to earlier marked nodes (else z would be evicted before y). The path Q joins two different supernodes during this iteration of the loop and hence has a subpath that also joins two supernodes and has no edges of G'. Thus by minimality of $|SP|$ we have $|SP| \le |Q| \le n(x) - n(y)$.

Now from Lemma 7 we have $n(x) \le |C|/2$, whence $|SP| \le |C|/2$. □

Lemma 9 *Procedure FindVineDecomp produces a vine decomposition $\mathcal{V}$ whose vines have value at least $R_2/4$, where R_2 is the number of reps of type 2. The number of nodes in $\mathcal{V} - G'$ is at most half the number of nodes in vines.*

Proof: Clearly the output of FindVineDecomp is a vine decomposition, $\mathcal{V}$. During each iteration through the loop, the number of (type 2) reps that become incorporated into a supernode is at most four times the value of the vine that is identified (this follows from Lemma 7, part (iii)). Hence the vines of $\mathcal{V}$ have value at least $R_2/4$.

The bound on the number of nodes of $\mathcal{V}$ follows from Lemma 8. □

Lastly we show how to use the vine decomposition $\mathcal{V}$ to construct either a large cycle or a vine decomposition on $k+1$ nodes, without losing much value.

Theorem 10 *Either G contains a vine decomposition $\mathcal{V}'$ on $k+1$ nodes whose value is $\Omega(R_2)$, or a cycle on more than k nodes with value $\Omega(R_2)$.*

Proof: Let $\mathcal{V}$ be the vine decomposition given by procedure FindVineDecomp. We have two cases.

Case 1: $\mathcal{V}$ has a vine $\mathcal{C}$ with $r(\mathcal{C}) \ge R_2/16$. Suppose $\mathcal{C}$ has length l, and assume w.l.o.g. that $\mathcal{C}$ is contained in P_1. Let α be the length of the shortest cycle in G containing $\mathcal{C}$. If $\alpha \le k+1$, the cycle can easily be extended to a vine decomposition with value at least $\log l$, while by Lemma 7 Part (iii), $r(\mathcal{C}) = O(\log l)$, so $R_2 = O(\log l)$.

Otherwise $\alpha > k+1$. In addition, $\alpha \le k(1 + 2^{-r(\mathcal{C})/4})$, since this is an upper bound on the length of the cycle containing SP, P_1, P_2 and some nodes inside supernodes (by Lemma 8). Hence by Theorem 4, Part 2, the shortest cycle containing $\mathcal{C}$ has value at least $\lfloor r(\mathcal{C})/8 \rfloor = \Omega(R_2)$.

Case 2: No single vine $\mathcal{C}$ has $r(\mathcal{C}) \ge R_2/16$.

Let j be the number of nodes in vines. By Lemma 8, $\mathcal{V}$ has at most $k + j/2$ nodes, and by Lemma 9, $\mathcal{V}$ has value at least $R_2/4$. Delete the vines one by one, in decreasing order of length, till no more than $k+1$ nodes remain in $\mathcal{V}$. At most $j/2$ nodes are removed prior to the final deletion of vine $\tilde{v}$ that brings the number of remaining nodes below $k+1$. Since they are deleted in increasing order of ratio of value to number of nodes, the vine decomposition remaining before the deletion of $\tilde{v}$ has value at least $R_2/8$. The further deletion of $\tilde{v}$ leaves value at least $R_2/16$, since $\tilde{v}$ has value at most $R_2/16$. Lastly, we can easily extend the tree so that $\mathcal{V}$ has exactly $k+1$ nodes. □

Summary

We summarize the results of this section:

Theorem 11 *For any graph G and memory size k, the algorithm FAR is strongly competitive, i.e.*

$$c_{FAR,k}(G) = O(c_k(G)).$$

Proof: Let g be the number of new nodes requested in the current phase. By Proposition 3, the adversary incurs at least $g/2$ faults for this phase, amortized. Suppose that R is the number of reps for the phase: $2R$ is an upper bound on the competitive ratio of FAR for this phase. But $\Omega(R)$ is also a lower bound on $c_k(G)$. Indeed, either there are many type 1 reps, so we can find a tree with value $\Omega(R)$ using Theorem 6, or there are many type 2 reps, so we can use Theorem 10 to construct a vine decomposition on $k+1$ nodes or a cycle whose value is $\Omega(R)$. In either case, Theorem 4 gives $c_k(G) = \Omega(R)$. □

3 Structured Program Graphs

The problem of devising competitive paging algorithms for directed access graphs is an important one. Unfortunately, in full generality this problem seems very difficult. There is, however, a restricted class of directed access graphs that is both important and tractable, which represents a subset of the stream of instruction references made by a structured program. We examine the class of *structured program graphs* (*spg*'s) as defined in the introduction.

We assume that at run time, the paging algorithm knows the spg (the implications of this assumption are discussed in Section 4). The information that is not available until run time is the number of times a particular loop will be executed or the particular branch an execution path will take. In other words, it is unknown until run time exactly which path through the access graph will represent the access sequence for a particular execution. Fortunately, our analysis is worst case over all paths through a particular access graph. The algorithm we introduce, called EVEN, is strongly competitive on all spgs. EVEN is a variation of an algorithm introduced in [2] called 2FAR.

Recall that spgs are recursively defined as follows: Every spg has a designated *start* node and a designated *stop* node. A single directed edge is an spg, where the node with out-degree one is the start node and the node with in-degree one is the stop node. More complex spg's can be constructed by applying the following rules.

- Two spg's G_1 and G_2 can be combined to get a new spg, by identifying the start node of G_1 and the stop node of G_2 (serial composition).
- Two spg's G_1 and G_2 can be combined to get a new spg by identifying the start nodes and identifying the stop nodes of G_1 and G_2 (branching statement).
- A node v in an spg can be identified with a node u of a directed cycle to yield a new spg (loop). We call v the *pivot* node for the cycle.

For each subgraph G' of an spg G (including the entire graph), we construct an undirected graph $U_{G'}$, called the underlying graph of G', that represents the relative embedding of the loops of G'. To distinguish between G' and U'_G, we call nodes in G' *nodes* and vertices in $U_{G'}$ *vertices.*

Definition 4 *Let G' be a subgraph of an spg G. The* **underlying graph** $U_{G'}$ **of** G' *has one vertex for each loop l in G' (called a loop vertex), and one vertex for each pivot node in G' that is in at least two loops (called a pivot vertex). Each pivot vertex is adjacent to all the loop vertices corresponding to loops containing that pivot node. With each vertex u in $U_{G'}$, we associate a set of nodes of G', denoted $V(u)$: If u is a pivot vertex, then $V(u)$ is the corresponding pivot node. If u is a loop vertex corresponding to loop l, then $V(u)$ is the set of non-pivot nodes in l, and pivot nodes in l that are in only one loop.*

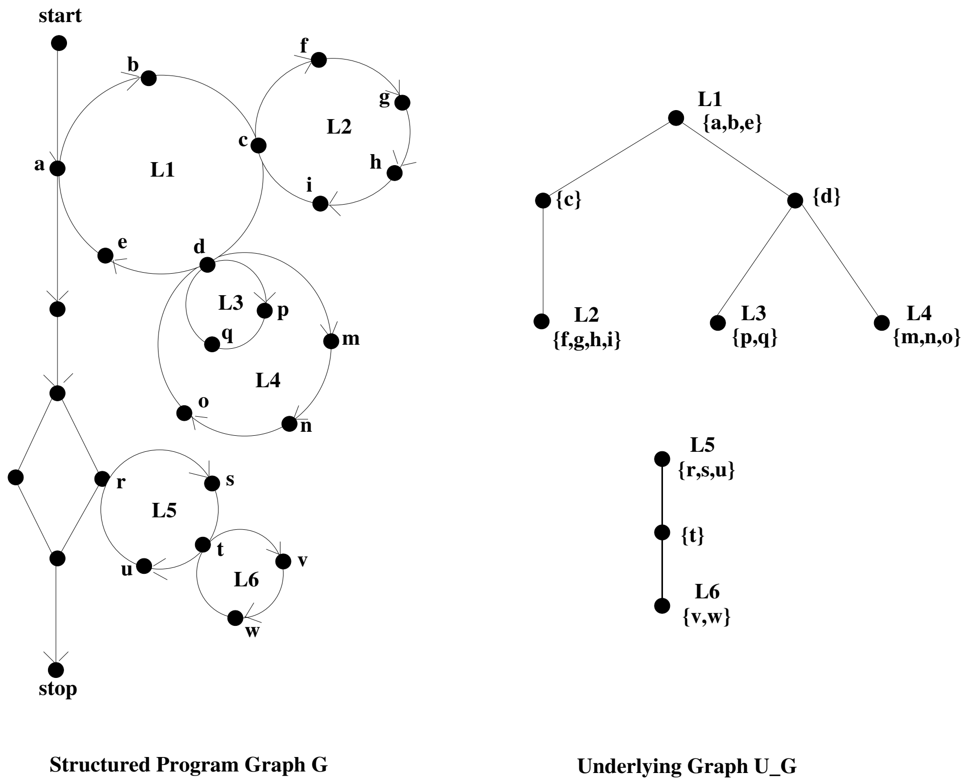

Figure 2: An spg and its underlying graph

Figure 2 shows an example of an spg G and its underlying graph U_G. For each vertex v in the underlying graph, the nodes in $V(v)$ are shown in curly braces. Vertices in U_G with the name of a loop next to them are loop vertices; the remaining vertices are pivot vertices.

A strongly connected subgraph F of G is a subgraph such that for any two nodes w and v in F, there is a path from w to v and a path from v to w that only uses nodes and edges from F. Notice that if F is a strongly connected subgraph of an spg G with more than one node and v is a vertex in F, then v is in a loop. In other words, $v \in V(u)$ for some $u \in U_G$.

Lemma 12 *The underlying graph U_F of a strongly connected subgraph F of G is a tree. The underlying graph U_G of an spg G is a forest.*

Proof: By the construction rules for spg's, a strongly connected subgraph F of G consists of a connected set of cycles. Suppose that U_F contains a cycle. Since two loop nodes or two pivot nodes are never adjacent, every loop node contained in the cycle must be adjacent to two pivot nodes in the cycle. Among the loops represented in the cycle, let l be the last one added to the spg according to the construction rules above. Suppose that v and w are the two pivot nodes adjacent to l in the cycle. Since l was added last, a loop containing v and a loop containing w were already in the spg at the time l was added. But by the construction rule for loops, l can only be adjacent to one of them, which contradicts the existence of this cycle. Therefore U_F must be tree.

Since U_G is the union of underlying graphs of maximal strongly connected subgraphs of G, and each of these is a tree, U_G must be a forest. $\square$

3.1 A Lower Bound

A vertex in U_F is said to be a *peripheral vertex* if its removal does not disconnect U_F. If u is a peripheral vertex in U_F, $V(u)$ is said to be a *peripheral set* in F. The *size* of a peripheral set is its cardinality, $|V(u)|$. Note that if a node is in a peripheral set of F, then it is contained in only one loop in F. For any strongly connected graph F on $k+g$ nodes, define $l(F,g)$ to be the number of peripheral sets in F that have size at least g. Whenever the value of g is clear from context, we say that a peripheral set is *large* if it has size at least g, and we abbreviate $l(F,g)$ by $l(F)$.

We need the following preliminary lemmas.

Lemma 13 *Let F be any $(k+g)$-node strongly connected subgraph of an spg. Let S be the set of peripheral sets in F. Then:*

1. *For any $s_1, s_2 \in S$, and for any nodes $x \in s_1$ and $y \in s_2$, there is a path in F from x to y that doesn't pass through any other peripheral set $s_3 \in S$, $s_3 \neq s_1, s_3 \neq s_2$.*

2. *For any y that is not in any peripheral set, and for any x, there is a path from x to y that doesn't contain nodes from any peripheral set (except perhaps for some nodes in s_1 such that $x \in s_1 \in S$).*

Proof: For the first part, consider the directed graph F' obtained from F by deleting, for each large peripheral set s_3 ($s_3 \neq s_1, s_3 \neq s_2$), all nodes in s_3. Since F' is still strongly connected, the desired path from x to y remains.

For the second part, consider the directed graph F' obtained from F by deleting, for each large peripheral set s ($s \neq s_1$ such that $x \in s_1$), all nodes in s. Since F' is still strongly connected, the desired path from x to y remains. □

The following theorem gives a lower bound on the competitive ratio achievable on an spg G. Let $\mathcal{H}_i(G)$ be the set of strongly connected subgraphs of G on i nodes. Let

$$comp(G) = \max_F \max\{(l(F)-1)/g, (H_{l(F)}-1)\},$$

where the maximum is taken over $F \in \mathcal{H}_{k+g}(G), k \geq g \geq 1$, and H_j is the j'th harmonic number.

Theorem 14 *If G is a structured program graph, then*

$$c_k(G) \geq comp(G).$$

Proof: Let F be any strongly connected subgraph of G on $k+g$ nodes, and let A be a paging algorithm. Let S be the set of large peripheral sets of F.

We now describe the adversary strategy. The adversary operates in phases (slightly different from the phases referred to elsewhere). We show that the adversary can force A to incur $g \cdot comp(G)$ faults in a phase while the adversary only incurs g faults in a phase. A phase starts with all of the adversary's holes on one set in S and the last request on a node in $\cup S$.

If the algorithm A ever maintains a hole on a node not contained in $\cup S$, or in a set in S that has already been requested in the phase, then the adversary requests a path to the hole without passing through any unrequested nodes in $\cup S$. Thus we can assume that all of A's holes are on unrequested nodes in $\cup S$. In this case, the adversary requests a path to the peripheral set in S with the most holes, such that the path followed doesn't hit any other peripheral sets in S. The phase ends when all but one of the large peripheral sets have been hit.

When the i^{th} peripheral set is hit, A incurs at least $\lceil g/(|S|-i+1)\rceil$ faults. Therefore, the algorithm incurs at least

$$\begin{aligned}\lceil g/|S|\rceil + \lceil g/(|S|-1)\rceil + \cdots + \lceil g/2\rceil &\geq \max(g(H_{|S|}-1), |S|-1) \\ &\geq g \cdot comp(G)\end{aligned}$$

faults. Meanwhile, the adversary services the sequence by initially moving all g of its holes to the peripheral set that will not have been hit at the end of the phase and thus incurs at most g faults during the phase.

□

3.2 The Algorithm EVEN

We now explain the operation of EVEN on spg's. EVEN works in two modes. The first mode is when the set of nodes occupied by the servers and the present request are not contained in a strongly connected component. In this case, EVEN services a fault by using any server which is on a node that is not reachable from the current request. In the second mode, all the servers are in a strongly connected component and EVEN works as a marking algorithm. A new phase begins either when EVEN switches from mode one to mode two or when $k+1$ nodes have been marked in the previous phase (during the second mode). Roughly, EVEN attempts to vacate servers in peripheral node-sets, and does so in a uniform way. A few definitions are necessary in order to rigorously describe the algorithm.

Let P be the set of nodes occupied by servers at the beginning of a phase. Let $\bar{P}$ be the smallest strongly connected subgraph that contains all of P.

Consider the underlying graph of $\bar{P}$, $U_{\bar{P}}$. Root $U_{\bar{P}}$ at the vertex whose node-set contains the last request of the previous phase. This vertex is unique. As noted earlier, each vertex u in the underlying graph of $\bar{P}$, $U_{\bar{P}}$, corresponds to a node-set in $\bar{P}$, $V(u)$. A node-set is *empty* if it does not have any servers on it. A *limb* $L = (l_1, \ldots, l_r)$ in $U_{\bar{P}}$ is a simple path in $U_{\bar{P}}$ such that l_1 is a peripheral vertex (leaf of $U_{\bar{P}}$) and l_{i+1} is the parent of l_i for $1 \leq i < r$. The *limb-set* $V(L)$ is the set of nodes corresponding to the vertices of the limb L (i.e. $\cup_{1\leq i\leq r} V(l_i)$). EVEN keeps track of the limbs, making sure that each limb has servers only on the nodes in $V(l_r)$, i.e. all but the last vertex on the limb is empty. The vertex l_r (the endpoint of the limb that is furthest from the leaf) is called the *active vertex*, and $V(l_r)$ is called the *active set*.

Limbs grow toward the root during the phase. At the beginning of a phase, each peripheral vertex is a limb. The peripheral set is the active set for that limb. Notice that because we chose $\bar{P}$ to be the minimum strongly connected subgraph that contains all nodes with servers, every peripheral set will contain at least one server. EVEN will always evacuate servers from the active set of some limb. A limb is said to *die* when any of the following conditions hold:

1. The active node-set empties and some node in the node-set of the parent vertex has been requested.

2. The active node-set empties and the parent vertex has another child that is not yet in a dead limb.

3. A node in the active set is requested.

A *live* limb is one that has not died. The *value* of a limb L is the number of nodes in the limb-set $V(L)$ that have been evacuated during the phase.

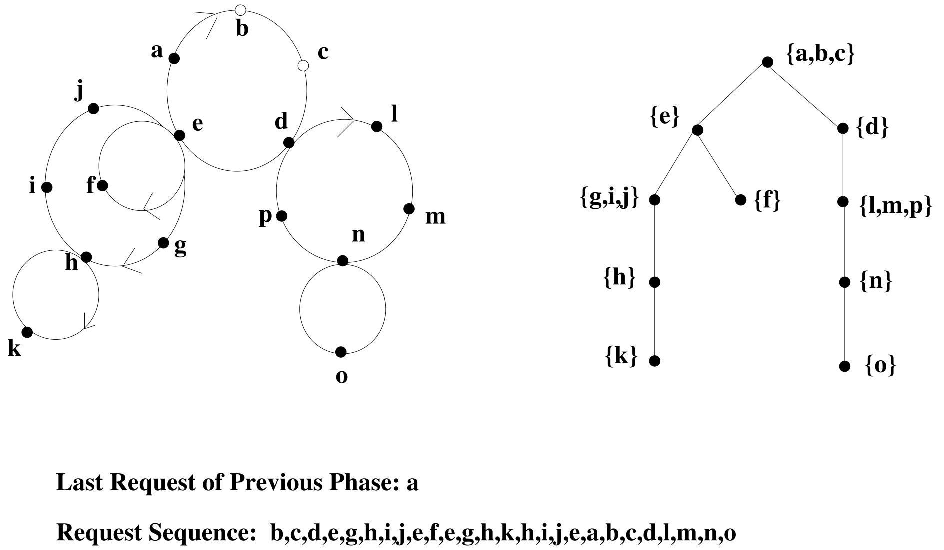

Figure 3: An example $\bar{P}$, $U_{\bar{P}}$ and request sequence

EVEN's policy for deciding which node to vacate when servicing a fault is determined as follows: pick the live limb of minimum value. Evacuate any node in the node-set of the active vertex. If the active vertex becomes empty then add the parent vertex to the limb (unless the limb dies for the reasons described above). The parent becomes the limb's active set. If there are no live limbs, then evacuate the unmarked node that is occupied by a server and is farthest from the current request.

Figure 3 shows an example of the operation of EVEN in mode 2. On the left side of the figure is the subgraph $\bar{P}$, and on the right side is the corresponding underlying graph $U_{\bar{P}}$. At the beginning of the phase all nodes are occupied by servers except for b and c, and the last request of the previous phase was at a. Initially there are three limbs, the peripheral vertices of the underlying graph: $\{k\}$,$\{f\}$ and $\{o\}$. The request to b results in the eviction of o, at which point, the parent vertex is added to the limb, resulting in the limb $(\{o\},\{n\})$, with active vertex $\{n\}$. The request to c results in the eviction of f, at which point we observe that the parent of $\{f\}$ ($\{e\}$) in the underlying graph has another child ($\{g,i,j\}$) which is not yet in a dead limb, and so by rule 2, the limb $\{f\}$ dies. At this point the two holes are at f and at o. Therefore, no further faults are incurred until the request to f at which point k is evicted. The parent vertex of $\{k\}$ (h) has already been requested, and so by rule 1, the limb $\{k\}$ dies. The next fault is at the request to k which results in the eviction of n, and the only remaining limb is extended to $(\{o\},\{n\},\{l,m,p\})$, with active set $\{l,m,p\}$. At this point, the two holes are at o and n. Finally, when l is requested, the limb dies by rule 3, since a node in the active set is requested.

In the next section, we bound the competitive ratio of EVEN.

3.3 The Upper Bound

We will need the following three lemmas.

Lemma 15 *Consider a phase of* EVEN . *If there are no live limbs, but there remain unmarked nodes that are occupied by servers, then all of the unmarked servers reside on the only simple path from the currently requested node to the first node that was requested in the phase.*

Proof: If no live limbs remain, then every loop with unmarked servers in $\bar{P}$ has a node which has been requested in the phase. Consider the set S of all nodes in $\bar{P}$ which have not been requested during the phase but are contained in cycles where some node has been requested. All unmarked servers reside on nodes in this set. Furthermore, since the set of requested nodes is contiguous, S forms a simple connected path in $U_{\bar{P}}$ from the most recently requested node to the first node requested in the phase. □

Lemma 16 *At any point in a phase of* EVEN*, if there are l live limbs, the value of each limb is bounded above by $\lceil g/l \rceil$, where g is the number of new nodes requested in the phase.*

Proof: The values of any two live limbs differ by at most 1. Furthermore, the total number of evacuated nodes in live limbs at any time is at most g. The lemma follows. □

Lemma 17 *Consider a phase of* EVEN *where $|\bar{P}| = k + g$. Let L be the set of limbs that grow in the phase. Then the limbs in L can be divided into two sets L_1 and L_2 and there is a strongly connected subgraph H of $\bar{P}$ of size $k + \alpha$, $1 \leq \alpha \leq g$ such that:*

1. *Every peripheral set in H has size at least α.*
2. $|L_1| \leq 2l(H, \alpha)$.
3. *The total number of nodes contained in limbs in L_2 and are evacuated during the phase is at most $2(g - \alpha)$.*

Proof:

We trim $\bar{P}$ by repeatedly applying the following procedure: Let R be the subgraph of $\bar{P}$ that remains. (initially $R = \bar{P}$). While R has $k + h$ nodes, but has peripheral sets of size less than h, apply the following rules: Pick a peripheral vertex r of U_R such that $V(r) < h$. Remove r from U_R and remove $V(r)$ from R. (Note that since every leaf of U_R is a loop vertex, R remains strongly connected.) If as a result of this operation there is a peripheral vertex v that is a pivot vertex, remove v from U_R and set $V(p(v))$ to be $V(p(v)) \cup V(v)$, where $p(v)$ is the parent of v in U_R. (Note that $p(v)$ is again a loop vertex.)

Call the resulting graph H. By the termination condition of the procedure, if H has $k + \alpha$ nodes, then every peripheral set has size at least α.

Figure 4 shows an example of the trimming operation. On top we see $\bar{P}$ and $U_{\bar{P}}$, with thick lines on the underlying graph representing the limbs. During the trimming process, loops a, b, c and d are trimmed. The resulting underlying graph, U_H is shown in the bottom left. Note that in the trimming process, limb l_4 was removed completely, whereas only part of limb l_2 was removed. Note that the intersection of l_2 with H does not contain a peripheral set of H.

Returning to the remaining two conditions of the lemma, if a limb in L has no intersection with H, then put it in L_2 and remove it from L. The total number of nodes in the limbs placed in L_2 so far is at most the total number of nodes removed from $\bar{P}$ which is in turn upper bounded by $g - \alpha$.

All remaining limbs in L intersect H. To account for these, create a limb-tree as follows: Create a special root node. In addition, for each limb, create a node in the limb tree. Limb l is the parent of limb l' if the highest vertex in l' is the child of a vertex in l in the underlying graph $U_{\bar{P}}$. Any remaining limbs without parents take the root to be their parent. Note that any leaf in the limb-tree corresponds to a limb l that contains a peripheral set in H. Other limbs may or may not contain peripheral sets of H. Figure 4 shows an example limb-tree.

Let L' be the set of nodes in the limb-tree whose parent has degree 2 (and is not the root). Then $|L'|$ is the number of degree 2 limb-nodes in the tree. Therefore, there are at least $|L| - |L'|$ non-degree 2 limb-nodes. But the average degree of nodes in a tree is less than two, and so at least

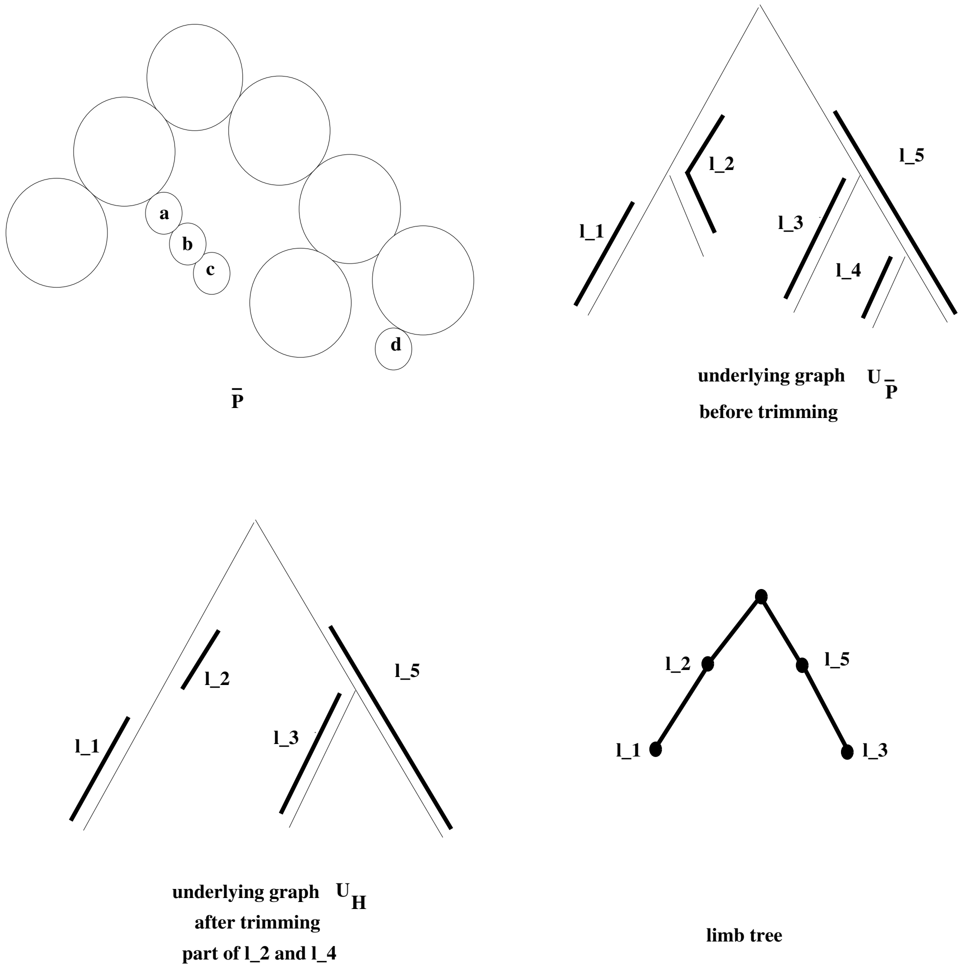

Figure 4: An example of the trimming process and limb-tree

$(|L| - |L'|)/2$ limbs in the tree are leaves and hence contain a peripheral set in H. We place the limbs in $L \setminus L'$ in L_1. Therefore, the total number of limbs in L_1 is at most $2l(H, \alpha)$ so far.

Consider a limb l in L'. Let $p(l)$ be the limb corresponding to the parent of l in the limb-tree.

If $p(l)$ contains a peripheral set of H, add l to L_1. This limb can be amortized against the peripheral set belonging to $p(l)$. Note that $p(l)$ is not a leaf of the limb-tree, and therefore is not charged against by L_1 limbs from $L \setminus L'$. ($p(l)$ may or may not end up in L_1 itself depending on whether it is in L' or not.) Consequently, the total number of limbs in L_1 remains at most $2l(H, \alpha)$.

If $p(l)$ does not contain a peripheral set in H, then place l in L_2. We just have to prove that the number of evacuated nodes in $p(l) \cap \{\bar{P} \setminus H\}$, which we denote by S, upper bounds the number of evacuated nodes in l. (A nodes is said to be *evacuated* if it was evacuated some time in the phase). Since l is the only child of $p(l)$, we are amortizing against the set S only once.

Let t be the highest vertex in l. Let T be the parent of t and t' the only sibling of t in $U_{\bar{P}}$. Since the limb $p(l)$ has only one child in the limb-tree and does not contain a peripheral set in H, t' and all of its descendants in $U_{\bar{P}}$ do not appear in U_H. That is, the set of nodes in $p(l)$ corresponding to t' and its descendants is exactly S. Recalling that the loop T is part of the limb $p(l)$ (otherwise $p(l)$ wouldn't be the parent of l in the limb-tree), it must be that l died before $p(l)$ died. This means that t became empty before t' became empty and so the number of evacuated nodes in l is at most $|S|$.

□

Theorem 18 *The algorithm* EVEN *is strongly competitive on the class of structured program graphs. In other words, for any structured program graph G, $c_{k,\mathrm{EVEN}}(G) = O(comp(G))$.*

Proof: Consider an spg G. If EVEN incurs a fault on node v when in mode one, then it is the first time that node v has been requested during the entire sequence. As a result, both EVEN and the optimal algorithm fault on the request to node v. Thus, when EVEN is in mode one, the number of faults that it incurs is equal to the number of faults the optimal algorithm incurs. We must now bound the number of faults incurred by EVEN in mode two.

Let P' be the set of nodes requested in the phase before the previous phase. Let P be the set of nodes requested in the previous phase and let C be the set of nodes requested in the current phase. If the algorithm just switched from mode one to mode two, then P is the set of nodes currently occupied by the servers, and P' are the k distinct nodes requested previous to the first request to any node in P.

Let g' be the number of new nodes requested in the last phase ($g' = |P - P'|$) and let g be the number of new nodes requested in this phase ($g = |C - P|$). If the number of faults EVEN incurs for an arbitrary phase is bounded by $O((g + g')c_k(G))$, then by Proposition 3, $c_{k,\mathrm{EVEN}}(G) = O(c_k(G))$.

If $P' \cap C = \emptyset$, then $g' + g \geq k$. In this case, EVEN incurs $O(g' + g)$ faults during the phase, since no marking algorithm incurs more than k faults in a phase. If $P' \cap C \neq \emptyset$, then there is a strongly connected component containing P, contained in $C \cup P \cup P'$. Indeed, there is a directed path from the first node requested in C to a node x in $P' \cap C$. (This path contains at most g nodes outside P.) Since $x \in P'$ there is a path from x to the first node requested in the previous phase. (This path contains at most g' nodes outside P.) Finally there is a path entirely inside P from the first node requested in the previous phase to the first node requested in the current phase. Therefore, if $\bar{P}$ is the smallest strongly connected subgraph containing P, $\bar{P}$ has at most $k + g + g'$ nodes.

Let L be the set of limbs in $\bar{P}$ that evolve in the phase. We now apply Lemma 17 to obtain a graph H of size $k + \alpha$ and a partition of L such that:

1. Every peripheral set in H has size at least α.

2. $|L_1| \leq 2l(H, \alpha)$.

3. The total number of nodes contained in limbs in L_2 that are evacuated during the phase is at most $2(g + g')$.

We can now show that the number of faults that EVEN incurs in the current phase is $O(g + g' + l(H) + gH_{l(H)})$. We count the number of faults by counting the number of nodes evacuated in $\bar{P}$. In fact, we only count the number of nodes in L that are evacuated. This sufficient because Lemma 15 implies that there are at most $g + g'$ faults in between the time that the last limb dies and the end of the phase. As long as there are live limbs, all evacuated nodes are contained in limbs. Thus, the number of nodes that are not in any limb in L and are evacuated during the phase is at most $g + g'$.

We now bound the number of nodes in L that are evacuated. The number of nodes in L_2 that are evacuated in the phase is at most $2(g + g')$ by condition 3 in Lemma 17.

The number of evacuated nodes that are on limbs in L_1 is just the sum over all limbs in L_1 of the value of that limb when it dies. Consider the i^{th} limb in L_1 that dies. By Lemma 16, the number of evacuated nodes on that limb is at most $\lceil g/(|L_1| - i + 1) \rceil$. Thus by condition 2 of Lemma 17, the total number of evacuated nodes in L_1 is at most

$$\sum_{j=1}^{2l(H)} \lceil (g/j) \rceil = O(l(H) + gH_{l(H)}).$$

We have shown that the number of faults that EVEN incurs in the current phase is $O(g + g' + l(H) + gH_{l(H)})$. Since $\alpha \leq g + g'$ and, by Theorem 14, $l(H)/\alpha + H_{l(H)} \leq c_k(G)$, the number of faults EVEN incurs in the phase is

$$O(g + g' + l(H) + gH_{l(H)}) = O\left((g + g')\left(\frac{l(H)}{\alpha} + H_{l(H)}\right)\right) = O((g + g')c_k(G)).$$

□

4 Limitations of the Model and Open Problems

There are four fundamental limitations to our work. The first is the fact that the model of structured program graphs that we use does not allow branching within loops.

The second limitation concerns our assumption that the paging algorithm knows the access graph. This is not realistic for several reasons. If the set of virtual pages accessed by a structured program and the data on those pages were static, life would be easy. In fact, the virtual pages associated to the data and instructions of a program typically fall into three categories.

- Static: At link time, program text (code) and global variables are assigned virtual addresses. These addresses remain fixed for the lifetime of the program.

- Heap: Dynamically allocated data are assigned virtual addresses at runtime, when the memory is allocated. As long as these data are not deallocated, their virtual addresses remain fixed. However, once this storage is reclaimed, these virtual addresses can be allocated to other data. Consequently, the data on virtual pages associated to the heap may change many times over the lifetime of the program.

- Stack: Variables local to procedures are assigned virtual addresses on the stack. Since the stack is growing and shrinking as the program runs, in accordance with the set of procedures being executed, a specific procedure's local variables may have several different virtual addresses over the runtime of the program.

Consequently, a given virtual address may store many different data items, and a given data item may reside in several different virtual addresses. This implies at the very least that one cannot directly associate program data with specific virtual pages.

Nonetheless, access graphs have well-defined semantics: An edge from virtual page p_1 to virtual page p_2 in an access graph means that at some point in the virtual address trace of the program, a reference to an address on p_2 immediately follows a reference to an address on p_1.

Since the access graph depends on the virtual address trace of the program, which in turn depends on the input data, it is a nontrivial problem to construct it. (In fact, in its most general form, the problem is undecidable since one must know if the program will halt in order to determine the access graph.) In practice, if one wishes the access graph to be accurate, one may need to execute or simulate the execution of the program.

This limitation raises several interesting open questions: Are there good approximations to the access graph that can be determined at compile time? Is there a strongly competitive algorithm that "learns" the access graph as more requests are seen? Also, are there good algorithms that only maintain a portion of the access graph, say, the subgraph generated by the pages in fast memory?

A third fundamental limitation of this model is that there is only a single pointer into the access graph at any time. For example, the spgs that we have studied only make sense in the context of the instructions of a program. Spg's do not adequately model both the instruction and the data references of a program. In order to model both, one would at least need two pointers into the access graph, one into a structured program portion of the graph, and another into an undirected or directed subgraph whose pages are storing program data. The request sequence would then consist of a walk on the graph, where the next request is always a neighbor of one of the two nodes pointed to. One could also imagine having more than two pointers, for example, if there were a collection of pointers into arrays, or other data structures in the program. It remains an interesting open question to generalize our results to the multiple pointer case.

The last fundamental limitation of our work is that the competitive analysis we employ is worst case. We optimize the worst case ratio between our algorithm and the optimal off-line, assuming that an adversary chooses the request sequence. Locality of reference is enforced by limiting the adversary's sequences to walks on the access graph. Nonetheless, walks on the access graph may bear little resemblance to the program address trace from which the access graph was generated. Therefore it is not obvious that optimizing the competitive ratio against such an adversary yields an algorithm that performs well in practice.

Recently, Karlin, Phillips and Raghavan attempted to address this problem by studying paging algorithms for the case where the request sequence forms a Markov chain. The Markov chain incorporates locality of reference into the sequence, while eliminating the whole notion of an adversary.

The theoretical work on paging with access graphs could be complemented very well by experimental studies. It would be interesting to evaluate whether there are approximations to access graphs that model realistic page request sequences well. It would also be extremely interesting to evaluate how well the competitive algorithms described in this paper perform in practice.

Another interesting direction for research relates to granularity. In the introduction, we mentioned that an undirected access graph might be a suitable model for a program that performs operations on a tree data structure, as long as the mapping of the tree nodes to pages of virtual memory represents a contraction of the tree. Similarly, spgs are good models of the flow of control among individual program statements. However, once the instructions are assigned virtual addresses, the underlying access graph may not look anything like an spg, since many instructions are assigned to a single virtual page.

For a fixed page replacement strategy, different assignments of virtual addresses to data and procedure entry points can result in vastly different page fault rates, since changing the layout changes the partition into pages. It would be very interesting to study techniques for laying out instructions and data in virtual memory, so that the partition into virtual pages results in a low page fault rate.

Finally, dealing with the most general definition of structured program graphs remains open, as well as two questions from [2]:

Open Question 1 *Design an algorithm that is strongly competitive on all structured program graphs. (I.e. allow branching within loops.)*

Open Question 2 *Show that for all G and k, $c_{\mathrm{LRU},k}(G) \leq c_{\mathrm{FIFO},k}(G)$.*

Open Question 3 *Is there a "universal" randomized algorithm that is close to optimal on every G? Is there a graph-theoretic lower bound on $c_k(G)$ for randomized algorithms against an oblivious adversary?*

References

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

[2] A. Borodin, S. Irani, P. Raghavan, and B. Schieber. Competitive Paging with Locality of Reference. In *Proc. 23rd Annual ACM Symposium on Theory of Computing*, pages 249–259, 1991.

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

[4] 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.

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

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

[7] 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.

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

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

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

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

[12] D. Cheriton and K. Harty. Application-controlled physical memory using external page-cache management. Technical Report draft, Department of Computer Science, Stanford University, 1991.

[13] D. McNamee and K. Armstrong. Extending the Mach external pager interface to accommodate user-level page replacement policies. Technical Report 90-09-05, Department of Computer Science and Engineering, University of Washington, 1990.