Experimental Studies of Access Graph Based Heuristics: Beating the LRU Standard?

Amos Fiat* Ziv Rosen*

Abstract

In this paper we devise new paging heuristics motivated by the access graph model of paging [2]. Unlike the access graph model [2, 7, 4] and the related Markov paging model [8] our heuristics are truly on-line in that we do not assume any prior knowledge of the program just about to be executed.

The Least Recently Used heuristic for paging is remarkably good, and is known experimentally to be superior to many of the suggested alternatives on real program traces [17]. Experiments we've performed suggest that our heuristics beat LRU fairly consistently, over a wide range of cache sizes and programs. The number of page faults can be as low as 1/2 the number of page faults for LRU and is on average 7 to 9 percent less than the number of faults for LRU. (Depending on how this average is computed.)

We have built a program tracer that gives the page access sequence for real program executions of 200 – 3,300 thousand page access requests, and our simulations are based on 25 of these program traces. Overall, we have performed several thousand such simulations. While we have no real evidence to suggest that the programs we've traced are typical in any sense, we have made use of an experimental "protocol" designed to avoid experimenter bias.

We strongly emphasize that our results are only preliminary and that much further work needs to be done.

1 Introduction

In virtual memory systems program memory is viewed as a large virtual address space, divided into fixed size pages. The paging problem is that of deciding what pages of the virtual memory space should be stored in fast memory and what pages should be stored at a lower level of storage. A page fault occurs when a reference is made to a page that is not in fast memory. In this case, some page must be removed from the fast memory so as to allow the new referenced page to be brought in. A *page replacement strategy* specifies which page to replace on a page fault.

Competitive analysis of on-line problems [16], is used in theoretical studies of on-line algorithms to determine how well the algorithm is doing. In this style of analysis, the performance of the online algorithm is compared to the performance of the off-line algorithm on every input.

Sleator and Tarjan showed that no deterministic paging algorithm can achieve a competitive ratio better than k, where k is the number of pages of fast memory. They also showed that algorithms such as First-In-First-Out (FIFO) and Least-Recently-Used (LRU) are k-competitive, and hence best possible in their model. We study the competitive ratio of our heuristics in section 5. We show that our heuristics have a competitive ratio of $\theta(k \log k)$, much worse than that of the other algorithms tested. This is not surprising for the competitive ratio determines how well an algorithm is doing against a worst case adversary and does not necessarily determine how well the algorithm is doing for real inputs.

In [17] Young gives the results of experimentation where LRU consistently beats the alternative algorithms tested, including the marking algorithm of [5]. This is interesting because the competitive ratio of the randomized marking algorithm against an oblivious adversary is $\log k$ whereas the competitive ratio of LRU is k. This may suggest that the competitive ratio may be the wrong criteria to consider.

Several recent experimental studies of alternatives to LRU have appeared in the literature [9, 11, 12, 15, 10]. [9, 11] consider the problem of caching ATM sessions for transmitting IP over ATM, [12, 15] consider disk caching for database disk buffering, where the goal of [15] is to minimize response time rather than the number of misses, [10] gives neural network based algorithms for cache line buffering. Many of the new algorithms studied

*Department of Computer Science, Tel Aviv University, Tel Aviv, Israel. E-mail: {fiat,rosen}@math.tau.ac.il.

are adaptive and try to reflect changing statistical parameterization of the sequence. In this context our algorithm is very similar although there is no explicit statistical assumption used. One of the authors of [9, 11] has run one of the variants of [11] on some of our traces, it seems that this gives very good results for some of the range of cache sizes but performs rather worse on others[14].

In this paper we consider again the question as to whether the ideas being considered in the context of competitive analysis are relevant in practice. We consider several heuristics, based on the access graph model of [2], that seemingly give excellent results.

LRU is superior to our heuristics on less than 20% of approximately 300 simulation runs (trace, cache size), LRU is comparable to our heuristics on another 20% of the simulations, and is inferior on the remainder. On average, considering all 300 simulation runs, and taking the average of the ratios obtained for each such simulation run, LRU has 7% more page faults. If we simply count the total number of page faults for our heuristics and the total number of page faults for LRU, the ratio between the two gives our heuristic an average 9% advantage (averaged over the different ratios between the cache size and working set).

We have written our own software tools to generate appropriate program traces and have devised an experimentation "protocol" to try to avoid or diminish experimenter bias. All of the simulations discussed in this paper were performed after the scheme parameters were fixed. This is to avoid the problems of experimenter bias and of "cooking" the results by "tuning" the parameters to fit the required outcome.

We use as a measure of performance the *observed competitive ratio*, *i.e.*, the ratio between the number of page faults for an algorithm divided by the number of page faults for an optimal algorithm for the sequence.

We have modified the access graph model in that we don't assume that the access graph is known in advance, we build the access graph dynamically over time. Thus, our algorithm is really on-line. Also, we introduce an element of "forgetfulness" not present in the original access graph formulation. We try to learn the access graph representing the locality of reference, and at the same time, we try to forget out dated information so as to respond to changing situations.

Within this framework of a dynamic access graph we have considered both directed and undirected graphs. We consider several dynamic graph based heuristics, all of whom are seemingly superior to LRU. Amongst the various heuristics we consider there is no clear winner.

Notation:

V — Vertex set of dynamic access graph, E — Edge set
$w(e)$ — Weight of edge $e \in E$. k — Cache Size
c — page sequence counter. α, β, γ — Parameters.
(In our implementation $\alpha = 0.8$, $\beta = 1.5$, $\gamma = 10$.)

Initialization:

Let v be the page at which execution begins.
$V = \{v\}$, $E = \emptyset$.
Set current pointer $P = v$.

Undirected Dynamic Graph Paging Algorithm:

Repeat

Let v be the page pointed to by P.
When next address is on page $u \neq v$:
 if $u \notin V$
  then add u to V
 Let $e = (u, v)$
 if $e \notin E$
  then add e to E, set $w(e) = 1$.
  else $w(e) = \alpha \cdot w(e)$.
 if $w(e) > 1$ then set $w(e) = 1$.

Increment c: $c = c + 1$.
if $c = 0 \pmod{\gamma k}$
 then for all $e \in E$ do $w(e) = \beta \cdot w(e)$.

if u is not in the cache,
 then if the cache has empty slots: page u in,
  otherwise evict the page in the cache furthest from v
  in the weighted graph (V, E, w) and then page u in.

Set P to point to u.

Figure 1: The undirected dynamic access graph paging algorithm.

Interestingly, we obtain good results even when using an undirected access graph. The use of undirected access graphs in the theoretical studies of [2, 7, 4] was viewed as a first step simply because the directed case was (and still is) not well understood. It was conjectured that the undirected access graph could be fine for paging on data structures but would not be a good model of program behavior (because programs do have a direction inherent within them). Still, our heuristic based on the undirected access graph approach seems to beat LRU.

It seems that "forgetfulness" is crucial, we have simulated the original access graph algorithm (FAR) [2] on an access graph that describes the true access pattern. This requires a two-pass approach and is *not* on-line. We have found that the two pass FAR algorithm is often worse than LRU on program traces, whereas our heuristics, modified

Advantage over LRU — Total Page Faults																		
Cache as Percentage of Working Set																		
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
4%	4%	10%	6%	4%	5%	5%	8%	11%	20%	15%	1%	2%	9%	12%	13%	6%	11%	25%

Table 1: The first line above gives the cache size as a percentage of the working set. The percentages shown in the table show the difference (in percentages) between page faults for LRU and for our heuristic. (LRU has more page faults). These values were computed by dividing the total number of page faults, summation over all relevant simulation runs, for LRU and our heuristic.

Advantage over LRU — Average Competitive Ratio																		
Cache as Percentage of Working Set																		
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95
5%	3%	7%	5%	6%	13%	8%	4%	5%	13%	10%	1%	3%	8%	10%	11%	5%	4%	15%

Table 2: This table is like Table 1 except that we compare the unweighted average of the observed competitive ratios of the relevant simulation runs. *I.e.*, the values above reflect the expected value of the difference in percentages between the number of page faults for LRU and our heuristic if one of the relevant simulation runs was chosen uniformly at random.

forgetful variants of FAR, are strictly superior. It should be mentioned that for some scenarios the 2 pass FAR algorithm was superior to the dynamic graph algorithm (and thus to LRU). This suggests that even within a specific program execution, different stages of the execution have entirely different characteristics and modifying ones behavior to deal with the shifting realities is very important. (Interestingly, the two-pass FAR algorithm seems to track LRU's behavior in the limited sense that the total number of page faults is only slightly higher for two-pass FAR over LRU).

A serious drawback to our heuristics is that they are considerably more difficult to implement than LRU. Not only in terms of computation but also in terms of the memory requirements. Partial answers to these objections may be

1. The first major goal is to beat LRU on real program traces. That this is at all possible is interesting.
2. It well may be the case that simplifications of our heuristics will also work well in practice.
3. The allowable complexity of the "page replacement strategy" depends on the relative costs of executing the strategy and having more faults. Clearly, complicated paging strategies may be totally impractical at the level of the high speed cache but may be important and useful in a cache that holds copies of files obtained from the Internet.
4. Our page replacement strategy is "lazy", *i.e.*, evicts a page only on a page fault. Most of the work can be done on a "miss", on a "hit" the only work required is to record it.

Our work can only be viewed as suggesting that it is worth while to continue the study of heuristics such as ours. This paper is a very preliminary study of such algorithms and no real conclusions can be drawn from it.

2 The Access Graph and Markov Paging Models

The Sleator-Tarjan results conflict with practical experience on paging in at least two ways. First, many paging algorithms have a competitive ratio of k, even though in practice LRU usually is better than the alternatives. Additionally, LRU usually incurs much less than k times the optimal number of faults, even though its competitive ratio is k.

Many programs exhibit *locality of reference* [1, 3, 16]. 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 an unweighted graph that has a vertex for every 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. The definition of the competitive ratio remains the same, except for this restriction on the request sequences. Let $c_A(G)$ denote the competitive ratio of an on-line algorithm A with k pages of fast memory on the access graph G. We denote by $c(G)$ the infimum (over on-line algorithms A with k pages of fast memory) of $c_A(G)$. Thus $c(G)$ is the best that any on-line algorithm can do.

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. A directed access graph might be a more suitable model for program flow.

Borodin *et al.* described a simple and natural algorithm called FAR, and proved that it achieves a competitive ratio within $O(\log k)$ of $c(G)$ for every graph G. Irani, Karlin, and Phillips [7] subsequently showed that in fact the competitive ratio of FAR is $O(c(G))$.

FAR is a "marking" algorithm [5], it works in phases. A phase ends when $k+1$ different pages have been accessed since the beginning of the phase (not including the $(k+1)$st page). Pages in the cache can be either marked or unmarked. Pages not in the cache are never marked. During a phase, if a page is accessed then it is marked, marked pages are never evicted from the cache. If a page fault occurs and all pages are marked then all marks are erased. Every unmarked page in the cache is associated with a distance (on the access graph G) to the set of currently marked pages. When a page fault occurs then the furthest unmarked page in the cache is ejected.

Borodin *et al.* also considered randomized algorithms for paging with locality of reference. A randomized algorithm A for paging on access graph G has competitive ratio $c^R(A,G)$ if $c^R(A,G)$ is the infimum of c such that for any input σ chosen by an oblivious adversary, and corresponding to a walk on G,

$$E(A(\sigma)) \leq cOPT(\sigma).$$

The competitive ratio on G, $c^R(G)$, is then the infimum over randomized algorithms A of $c^R(A,G)$.

Fiat and Karlin [4] give a universal randomized algorithm for undirected access graphs, *i.e.*, an algorithm that obtains the best possible randomized competitive ratio against an oblivious adversary, up to a constant factor.

No similar results are known for the case of directed graphs, but a specific family of directed graphs have been dealt with in [2].

A different approach was taken by Karlin, Phillips, and Raghavan in [8]. They consider a paging strategy where the input to the paging algorithm is a Markov chain where p_{ij} gives the probability that page j is referenced just after page i. They give a paging algorithm that is within a constant of optimal when the page accesses are generated according to a Markov chain.

Both the access graph model and the Markov paging model require that the paging algorithm have some pre-knowledge of the upcoming page access sequence. In the access graph model this is the access graph, in the Markov paging model this is the Markov chain.

3 Experimentation

To generate program traces we built our own "trace" program, this program works similarly to a debugger, it places a trap instruction at the next branch instruction, captures this, and thus follows the program counter throughout all changes.

Our work has two serious limitations:

- We only consider program access, not access to data.
- We consider the program as though it were running alone in a stand-alone computer, not in a multi-tasking environment.

It is our conjecture that our heuristics would actually work better if these two elements were added. This would require some modification to the algorithm to deal with multiple pointers (as done in the theoretical setting in [4]).

A detailed description of the results of the various simulation runs is presented in the results section below. Summary results include unweighted averages of the observed competitive ratios for a given fraction of the working set (see Table 2). We also consider the ratio between the total number of page faults for a given fraction of the working set, over all traces (see Table 1).

Program Name	Version	Working Set Size
latex	C 3.14t3	80
gnuplot	3.50.1.17	85
grep	Sun release 4.1	23
find	Sun release 4.1	35
gunzip	1.2.4	31
gzip	1.2.4	33
paging	Sun release 4.1	27

Table 3: Traced programs.

3.1 Experimental Protocol. It is not clear how much effect the various algorithm parameters have on the overall behavior of the algorithm. A promising feature of our algorithms is that they are relatively insensitive to small changes in the parameters.

Nonetheless, tuning the algorithm parameters on the actual input used to obtain experimental results seems to be problematic. The validity of such results may be doubtful. Thus, we have fixed the algorithm and its parameters and run simulations on traces of entirely new programs. Although we believe that some further modifications may prove to be worthwhile, we have overcome the urge to make further modifications in the algorithm because this would require entirely new inputs by protocol.

3.2 The Page Tracer. In order to generate a trace of the program counter (PC) of a given process's execution we built a utility called "pt" (program-tracer). The computer hardware platform used was Sparc-10 with the Sun4 CPU architecture. The operating system used was the Unix based SunOS.

The purpose of the program tracer utility is to execute any Unix process, and while doing so, record a list of all pages accessed.

The Sun4 architecture supports *trapping* the CPU execution, and a designated machine instruction, TRAP, is provided for this purpose. When executing the TRAP instruction the CPU stops the execution flow and continues it from another memory location. The SunOS/Unix operating system provides tools by which a process may control the execution of another process, and examine and change its core image. The primary use of these tools is for the implementation of breakpoint debugging.

Our program tracer is a "shell" process that executes the process which we want to trace. The tracing shell process installs TRAP instructions in the traced process's core image, and when the traced process executes the TRAP instruction it stops immediately, and the control returns to the shell process.

The SunOS/Unix system indicates that the last instruction executed was a TRAP by a special Unix signal, SIGTRAP. Our program tracer installs the TRAP instruction at the next instruction to be executed after the instruction that the traced process is just about to execute. After installing the TRAP instruction the shell process lets the traced process execute this single instruction. During this time the shell process itself is in a *wait* state.

If the upcoming instruction is a branch instruction, then TRAP instructions are installed in all candidate next addresses. The Sun4 instruction set provides four types of branch instructions, each of which requires somewhat different computations.

After having noted the PC register value when the traced process stopped, the shell process returns the machine instructions that were modified to their original state.

The pt program is invoked from the command line and it gets as arguments the name and the argument list of the program to be traced. Another argument that pt requires is a *page size*. Since the Unix operating system runs its processes in a virtual memory environment, the PC register values that are recorded represent virtual memory locations. The page size argument specifies the size of the memory pages in our simulated machine. The program tracer uses the page size to output a list of accessed pages rather than virtual addresses.

In all our experiments, the page size argument was set to 4096 bytes, the true page size for the system in use.

We used the output file generated by the program tracer as an input to the programs that simulate paging strategies. Having the "page size" parameter set to 4096 bytes, we obtained trace files of between 200 and 3000 thousand pages.

A current limitation of our pt tracer utility is that it cannot deal with multiple processes.

All our traces, the pt program tracer utility, and software for simulating paging algorithms is available via www from www.math.tau.ac.il/~rosen. We strongly encourage anyone who performs simulations on our traces or makes use of these tools to inform us of the results. We give suggestions for additional heuristics to be tested in the final section of this paper.

4 Access Graph based Heuristics

Our heuristics share in common a process that builds a dynamic weighted access graph.

4.1 Building the Dynamic Access Graph.

We build *weighted* dynamic access graphs. Every page of the address space is associated with a vertex of the graph.

When an edge is crossed many times (the two pages associated with the endpoints are requested in succession) the edge weight goes down. Over time, all edge weights are increased. The intuition behind our construction is that pages whose vertices are close in the graph are strongly related and may be requested in close succession, pages whose vertices are far apart are less likely to be requested in close succession. We introduce the element of "forgetfulness" by periodically increasing the distance between all pairs of vertices.

4.1.1 The Undirected Dynamic Access Graph. Every page of the address space is associated with a vertex of the graph. Initially, there are no edges. An edge is created between two vertices on the first time two pages are requested in

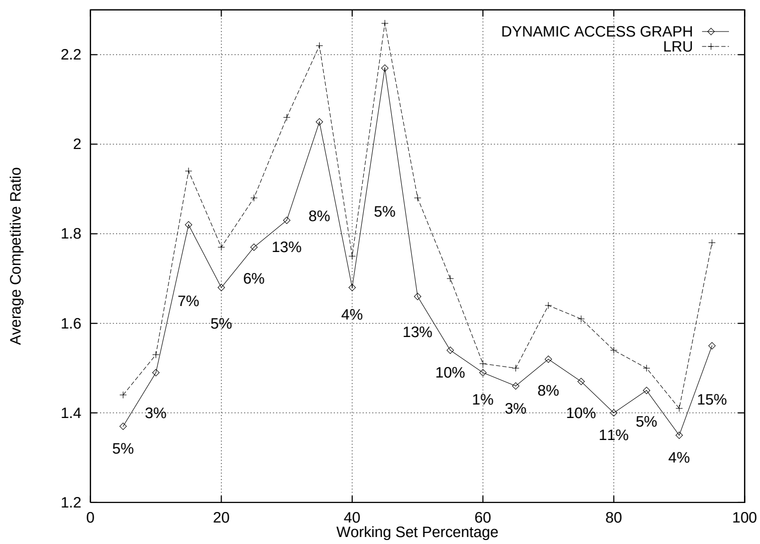

Figure 2: Observed competitive ratio comparison between the undirected dynamic access graph algorithm of figure 1 and LRU. The x axis gives the percentage of the cache size as a function of the working set. Every point on the graph was obtained by taking an unweighted average of the competitive ratios for the same x value on all 25 traces. The percentages in the figure represent by what fraction LRU's observed competitive ratio is higher.

succession. The weight associated with this edge is one.

If an edge is traversed, (the two pages are requested successively), the weight of the edge goes down by a constant factor α ($\alpha = 0.8$). However, if this new weight is more than one then the new weight of the edge is one.

Over time, the weight of all edges goes up, *i.e.*, multiplied by a constant factor β ($\beta = 1.5$) every $O(k) = \gamma \cdot k$ page accesses ($\gamma = 10$).

The undirected graph construction algorithm and the associated paging strategy can be seen in figure 1.

4.1.2 The Directed Dynamic Access Graph. The construction of the directed dynamic access graph is the same as above, only we deal with directed arcs rather than undirected edges.

4.2 Variants of FAR on the Dynamic Access Graph. We consider a variant of FAR on dynamic access graphs where the page to be evicted on a page fault is the furthest page from the *current* page, accessed just before the page fault. Thus, our algorithm is not a marking algorithm.

As can be seen from the experimental evidence, this variant is seemingly superior to the off-line version of FAR when run on the true access graph (see figure 4).

4.3 Results. Table 3 contains the list of programs for which we generated traces, which were used as inputs to the simulated paging algorithms. A total of 25 traces were generated subsequently to fixing the scheme parameters. These traces are described in table 4.

For all of these traces, simulations were performed for the various algorithms on a range of cache sizes. About 300 different simulation runs were performed. The optimal number of page faults attainable for these test instances is given in table 5. The observed competitive ratio of the undirected access graph heuristic for these same test instances in given in table 6. The observed competitive ratio of LRU on these same test instances, given as the difference in percentages from the observed competitive ratio of the undirected dynamic access graph heuristic, is presented in table 7. While they are not presented herein, the same tables were computed for several other algorithms and the average observed competitive ratios, as a function of the fraction of the working set size, are presented graphically in figure 3.

For different programs, the absolute cache size translates to a different fraction of the working set. Thus, to obtain average results over several traces reflecting different program executions we have normalized the results as a fraction of the

Trace no.	Program Name	Input File	Trace Length
1	gnuplot	airfoil.dem	982843
2	gnuplot	animate.dem	1270694
3	gnuplot	polar.dem	1626387
4	gnuplot	bivariat.dem	974130
5	gnuplot	contours.dem	1261116
6	gnuplot	world.dem	977748
7	latex	gpcard.tex	1732163
8	latex	refcard.tex	972917
9	latex	xfig.1.02.tex	688059
10	latex	xdvi.18f.tex	2132159
11	latex	dvipsk-5.58f.tex	1278003
12	latex	Report.tex	888055
13	gunzip	pt.latex.out.1.gz	309518
14	gunzip	gcc-2.6.tar.gz	409514
15	gzip	pt.latex.out.1	25738
16	gzip	gcc-2.6.tar	48489
17	paging		363797
18	paging		3340855
19	paging		463809
20	paging		3000458
21	grep	"Page"	1932966
22	grep	"Integer"	2076354
23	grep	"Float"	1930525
24	grep	"Char"	2069211
25	find	"stdio.h"	1991970

Table 4: Traces generated and the programs and inputs used to generate them. Traces 17–20 are of the Unix paging utility whose input is the system environment itself. Traces 21–24 are of the grep utility on a large directory.

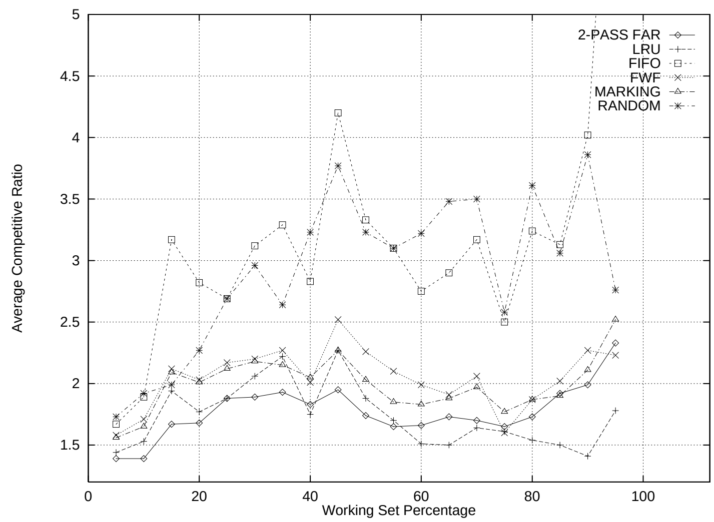

Figure 3: Average observed competitive ratio comparison. The 2-pass FAR algorithm is an implementation of the FAR algorithm of Borodin *et. al.* ([2]), on the true access graph of the trace. FWF is a deterministic phase based algorithm from Sleator and Tarjan ([16]), the marking algorithm is the randomized algorithm of Fiat *et. al.* ([5]), random simply selects a random page to evict. The x axis represents the fraction of the working set. The points of the different graphs are computed by taking the unweighted average of all the observed competitive ratios on all traces with a cache size equal to the appropriate fraction of the working set.

working set in figures 2 and 3. Figure 2 gives the average observed competitive ratio for LRU and out dynamic undirected access graph heuristic. Figure 3 gives the same results for several other algorithms.

5 Competitive Analysis of the Undirected Dynamic Graph Algorithm

In this section we show that a slight variant of the dynamic graph algorithm presented above has a competitive ratio of $\theta(k \log k)$. The only difference between the algorithms is that we do not allow the edges to shrink to less than $1/k$ in weight. We have tested this variant experimentally on all the traces and it does not differ in its behavior when compared to the original dynamic graph algorithm as described in Figure 1. Tables 1 and 2 are virtually unchanged when computed for this variant. We don't present this variant as our algorithm because of the experimental protocol discussed above.

A competitive ratio of $\theta(k \log k)$ is somewhat disconcerting because the competitive ratio of the other algorithms considered is k. However, one must realize that the competitive ratio optimizes for a worst case adversary over all possible event sequences. We hope that in the future it may be possible to explain the experimental results of this paper by considering an appropriately restricted adversary.

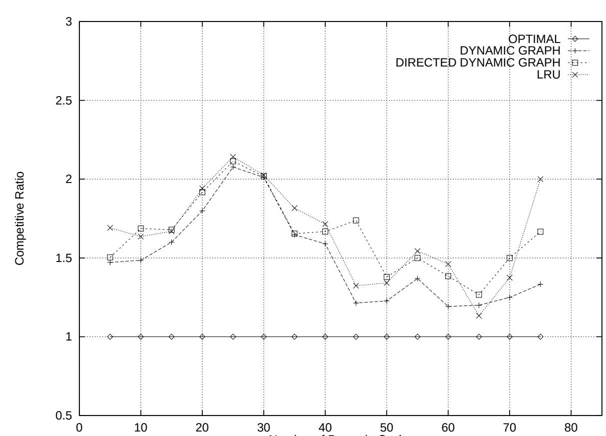

Figure 4: Observed competitive ratio comparison between the directed dynamic access graph based algorithm and the undirected version. Obtained from trace number 7.

Trace no.	Cache Size															
	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
1	191855	85199	16586	311	172	101	69	56	46	36	29	24	19	14	9	4
3	296500	131044	52038	8833	174	98	78	63	51	43	38	33	28	23	18	13
4	190293	84550	16440	323	179	112	84	67	52	41	31	24	19	14	9	4
5	201804	81137	21990	332	173	93	64	51	41	31	21	11	5	0		
6	190616	84629	16455	327	182	115	86	68	53	42	32	24	19	14	9	4
7	227872	46560	14444	7033	3515	1654	680	334	145	79	46	26	15	8	3	0
8	156731	44447	19245	9406	4489	1702	622	198	113	57	34	19	14	9	4	0
9	91856	18008	6290	2950	1354	703	339	147	87	53	36	23	13	6	1	0
10	285361	55617	17301	8326	4096	1975	863	442	213	124	75	45	29	16	6	0
11	209893	57414	24739	12176	5805	2062	725	268	157	93	63	43	28	18	8	0
12	124937	21580	7299	3368	1546	810	402	183	112	69	47	32	22	12	2	0
Trace no.	Cache Size															
	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33
13	87742	289	165	93	61	38	25	19	14	10	8	6	4	2	0	
14	115433	499	293	166	106	68	45	33	25	19	14	10	6	2	0	
15	452	255	145	82	53	35	24	19	15	11	9	7	5	3	1	0
16	590	330	184	104	68	46	31	23	17	11	9	7	5	3	1	0
17	90403	60232	36139	12088	56	38	26	16	9	6	4	2	0			
18	95034	63036	37906	12915	199	30	20	10	3	0						
19	118105	78671	47199	15790	79	52	37	23	13	9	5	2	0			
20	93969	62313	37467	12746	199	30	20	10	3	0						
21	20485	12731	6527	822	263	39	21	5	3	1	0					
22	13087	8262	4373	839	276	39	21	5	3	1	0					
23	20487	12732	6528	825	268	45	25	7	3	1	0					
24	13088	8266	4379	846	282	44	25	7	3	1	0					
25	918058	573358	358724	241130	147398	75747	7940	49	26	18	12	9	7	5	3	1

Table 5: Number of page faults for the optimal algorithm on the various traces and cache sizes considered.

Trace no	Cache Size															
	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
1	1.20	1.32	2.26	1.71	1.68	1.32	1.59	1.45	1.59	1.22	1.24	1.17	1.21	1.29	1.11	1.00
3	1.23	1.48	1.89	2.38	1.59	1.42	1.45	1.44	1.73	1.56	1.29	1.21	1.21	1.13	1.06	1.08
4	1.15	1.28	2.98	1.63	1.64	1.46	1.40	1.39	1.52	1.66	1.52	1.79	1.63	1.93	1.22	1.00
5	1.19	1.35	1.67	1.63	1.39	1.40	1.52	1.49	1.37	1.19	1.43	1.73	2.20			
6	1.20	1.16	2.97	1.53	1.76	1.46	1.40	1.13	1.36	1.45	1.53	1.67	1.84	1.93	1.56	1.00
7	1.47	1.49	1.60	1.80	2.08	2.01	1.65	1.59	1.21	1.23	1.37	1.19	1.20	1.25	1.33	
8	1.56	1.55	1.80	1.91	2.17	2.46	2.04	1.57	1.36	1.21	1.18	1.42	1.21	1.11	1.00	
9	1.51	1.64	1.83	1.96	2.04	1.82	2.21	1.61	1.37	1.79	1.39	1.26	1.08	1.33	1.00	
10	1.71	1.65	1.61	1.77	1.86	1.62	1.73	1.78	1.46	1.41	1.49	1.36	1.31	1.38	1.17	
11	1.42	1.56	1.71	1.88	1.78	2.38	2.17	1.53	1.41	1.35	1.32	1.19	1.68	1.89	2.50	
12	1.74	1.66	1.64	1.94	1.97	1.88	2.18	1.84	1.54	1.29	1.51	1.38	1.09	1.58	2.50	
Trace no	Cache Size															
	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33
13	1.93	1.32	1.27	1.38	1.39	1.24	1.20	1.26	1.36	1.60	1.62	2.00	1.75	1.00		
14	1.99	1.57	1.76	1.63	1.71	1.85	1.56	1.58	1.64	2.00	2.07	1.90	1.67	3.50		
15	1.30	1.25	1.45	1.65	1.85	2.03	1.83	1.42	1.27	1.36	1.56	1.71	1.40	1.00	1.00	
16	1.31	1.28	1.74	1.68	1.62	1.87	2.03	1.70	1.76	1.55	1.44	1.71	1.40	1.00	1.00	
17	1.55	1.54	2.42	3.65	1.45	1.50	1.58	2.06	1.78	1.17	1.25	1.00				
18	1.54	1.49	2.48	4.13	2.28	1.77	1.55	1.90	2.00							
19	1.55	1.49	2.32	4.26	1.63	1.87	1.84	2.00	1.54	1.44	2.20	3.50				
20	1.55	1.49	2.48	4.50	2.37	1.70	1.80	1.90	2.00							
21	1.10	1.49	1.71	1.60	2.86	1.67	2.24	1.80	1.00	1.00						
22	1.07	1.69	1.76	1.37	1.37	1.36	1.43	1.40	1.00	1.00						
23	1.11	1.40	2.42	1.78	2.53	2.27	2.24	1.86	2.33	3.00						
24	1.08	1.55	1.80	1.67	2.61	2.07	2.20	1.43	1.00	1.00						
25	1.20	1.19	1.34	1.76	2.07	3.09	2.02	1.80	1.19	1.17	1.25	1.22	1.14	1.20	1.00	1.00

Table 6: Observed competitive ratios for the undirected dynamic access graph based strategy on the various traces and cache sizes. *I.e.*, to obtain the number of page faults for a given trace and cache size where the observed competitive ratio in the table above is x, multiply the matching entry in table 5 by x.

Trace	Cache Size															
no	5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80
1	1.7	15.9	40.7	4.7	7.1	15.9	-0.6	3.4	-6.9	16.4	13.7	28.2	21.5	5.4	0.0	0.0
3	3.3	4.1	14.3	12.6	17.0	16.2	-2.1	-0.7	-16.2	-13.5	6.2	9.9	3.3	0.0	0.0	0.0
4	6.1	19.5	6.7	9.8	9.8	0.0	0.0	-1.4	1.3	-1.8	19.1	14.0	22.7	3.6	9.0	0.0
5	9.2	0.7	31.7	9.8	24.5	1.4	0.7	0.0	-2.2	8.4	13.3	26.0	54.5			
6	1.7	31.9	7.1	17.0	2.3	0.0	1.4	24.8	15.4	13.1	18.3	26.9	14.7	7.3	-7.7	25.0
7	15.0	10.1	4.4	7.8	2.9	1.0	10.3	8.2	9.1	8.9	12.4	22.7	-5.8	10.4	50.4	
8	0.0	2.6	0.0	0.5	0.9	7.3	2.5	-5.1	-3.7	1.7	11.9	3.5	-11.6	0.0	0.0	
9	11.9	1.8	-10.4	0.0	-1.0	0.0	-0.9	2.5	6.6	-21.8	-2.2	0.0	6.5	12.8	0.0	
10	0.0	0.6	3.7	8.5	12.9	22.2	3.5	-2.2	-4.1	3.5	20.1	19.1	5.3	4.3	0.0	
11	12.0	1.9	4.7	2.1	24.7	16.8	-6.9	0.0	0.0	0.0	14.4	34.5	1.8	14.8	20.0	
12	0.6	1.8	1.2	0.0	0.5	-3.7	-1.8	-7.6	1.9	22.5	12.6	6.5	4.6	-5.1	40.0	
Trace	Cache Size															
no	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33
13	3.1	-1.5	33.1	13.8	16.5	48.4	43.3	16.7	5.1	6.2	16.0	-16.5	-28.6	0.0		
14	0.0	-16.6	-2.3	-0.6	-0.6	1.1	14.1	1.9	-2.4	-5.5	10.6	21.1	39.5	14.3		
15	0.8	5.6	21.4	1.2	-5.4	-5.9	-4.4	0.0	4.7	14.0	7.1	8.8	14.3	0.0	0.0	
16	0.0	3.9	1.7	-2.4	3.7	-1.1	-6.4	-2.9	-2.8	0.0	16.0	8.8	14.3	0.0	0.0	
17	0.0	-2.6	3.3	36.2	3.4	14.0	19.0	2.9	0.0	13.7	0.0	0.0				
18	0.6	0.0	0.0	17.9	22.4	-16.9	12.9	5.3	16.5							
19	0.6	0.7	7.8	16.7	-9.2	-9.6	-1.6	0.0	9.7	8.3	0.0	0.0				
20	0.0	0.0	0.0	8.4	17.7	-13.5	-2.8	5.3	16.5							
21	0.9	7.4	59.6	0.0	15.7	1.2	0.0	-11.1	0.0	0.0						
22	3.7	-7.1	47.2	21.9	159.1	24.3	53.1	14.3	0.0	0.0						
23	0.0	14.3	12.8	-10.1	28.9	-25.6	0.0	15.1	14.6	33.3						
24	2.8	1.3	43.3	-0.6	34.5	-14.5	-1.8	9.8	0.0	0.0						
25	0.8	15.1	9.0	6.2	0.0	0.0	0.0	7.8	42.0	0.0	13.6	0.0	0.0	0.0	0.0	0.0

Table 7: Comparison of the observed competitive ratios for the undirected access graph based strategy and LRU. The difference is given as a percentage of the undirected access graph strategy observed competitive ratio. *I.e.*, to obtain the observed competitive ratio for LRU on a given trace and cache size, where the table entry above is x, take the matching entry from table 6 and add x%. Note that x may be negative which indicates that the observed competitive ratio for LRU is smaller than that of the undirected access graph based strategy.

5.1 The upper bound. Consider a sequence of requests, σ, resulting in $2k(\log_\beta k + 1)$ faults for the dynamic graph algorithm. If $k+1$ different pages are requested during this sequence then we're done, we can associate a cost of $\Omega(1)$ to the adversary using the phase analysis of [5].

Thus, only $i \leq k$ different pages are requested in σ. At least one page, p, must have been evicted twice over the last $k+1$ faults. On the last eviction of p, the distance between the current pointer and p depends on the number of requests between the previous request to p and the last eviction of p. If there were $\gamma k w$ requests in this interval then this distance is at most $k\beta^w$ where $\beta = 1.5$ is the constant by which we multiply the edge weights every γk requests.

Consider a page q not requested during σ. At the last eviction of p, the distance between the current pointer and q was at least $o \cdot \beta^{w+2\log k}$, where o was the weight of the last edge along this min weight path before σ was processed.

As $o \geq 1/k$ we get that the distance from the current pointer to q, at the last eviction of p, was at least $k\beta^w$, contradicting the eviction of p.

5.2 The lower bound. We now show that the competitive ratio of the undirected dynamic graph algorithm is no better than $\Omega(k \log k)$.

Consider the following scenario: A graph with $k+1$ vertices, vertices $r_1, r_2, \ldots, r_{k/2-3}$, connected to vertex R with edges of weight $O(1)$, vertices $l_1, l_2, \ldots, l_{k/2-2}$, connected to vertex L with edges of weight $\theta(1)$, vertex R is connected to vertex C_R with an edge of weight $\theta(1)$, vertex L is connected to vertex C_L with an edge of weight $\theta(1)$, C is connected to both C_R and C_L with edges of weight $\theta(1)$. Additionally, there is another vertex X connected to vertex C with an edge of weight $1/k$.

It is not hard to construct such a graph, successively request X and C many times, then request $\sigma = R, r_1, R, r_2, \ldots R, r_{k/2-3}, R$, followed by C_R, C, C_L, and then $\tau = L, l_1, L, l_2, L, l_3, L, \ldots, l_{k/2-2}, L$. The point is that there will be a page fault when $l_{k/2-2}$ is requested, but one of the pages r_i will be ejected. Then, the adversary will generate the sequence C, r_i, giving rise to a new page fault and some l_j page will be ejected. This will continue until the weight of the link XC becomes sufficiently high, this takes $k \log k$ requests, during which the algorithm has $\Omega(k \log k)$ faults. The adversary will evict the page at X immediately and has only two faults for this sequence. (We assume that initially the k pages of the r's , R, the C's, L and the l's are in the cache, and can repeat this process many times).

6 Future Work

We plan to continue this study in several directions. First, we would like to prove something about the relative competitiveness of the dynamic graph algorithms and the FAR algorithm given that the sequence is actually generated from a fixed access graph.

Variants of the algorithms described here may prove to be better. One idea not currently implemented is to try not to page out many pages that are close to one another in the dynamic access graph. *I.e.*, spread them about so that they are simultaneously far from the current pointer and (relatively) far from each other.

We hope to study the [4] algorithm on dynamic graphs, and to consider variants of this algorithm as well. Randomization actually helps in practice as can be seen from the relative performance of Flush-When-Full and Marking. We hope to introduce more efficient randomized heuristics.

We have some results on running FAR itself on a dynamic access graph, these are only preliminary but it seems that FAR on a dynamic access graph is very close to LRU and inferior to our algorithm. The reason for this seems to be that the marking process, while tuned towards a worst case adversary, actually prevents the algorithm from evicting pages that are likely not to be use.

We would also like to consider very simple variants of the dynamic graph algorithms. Hopefully, some variant sufficiently simple so as to be useful in practice. We note that branch prediction is performed in hardware [6].

Finally, we would also like to consider real traces of communications oriented caching, where algorithms such as those described in this paper may prove to be both implementable and cost effective.

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 and A.R. Karlin, Randomized and Multipointer Paging with Locality of Reference, In *Proc. 27th Annual ACM Symposium on Theory of Computing*, pages 626–634, 1995.

[5] A. Fiat, R. Karp, M. Luby, L. McGeoch, D. Sleator, and N. Young. Competitive paging algorithms *Journal of Algorithms*, 12(4):685–699, 1991.

[6] J.L. Hennessy and D. Patterson, Computer Architecture A Quantitative Approach, 1st edition, 1990, Morgan Kaufmann.

[7] S. Irani, A.R. Karlin and S. Phillips. Strongly competitive algorithms for paging with locality of reference. In *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms*, 1992.

[8] A. R. Karlin, S. Phillips, and P. Raghavan. Markov paging. In *Proceedings of 33rd Annual Symposium on Foundations of Computer Science*, 1992.

[9] S. Keshav, C. Lund, S. Phillips, N. Reingold, and H. Saran, An Empirical Evaluation of Virtual Circuit Hold Time Policies in IP-over-ATM Networks. IEEE Journal on Selected Areas in Communications, Vol. 13, No. 8, October 1995, pp. 1371–1380.

[10] H. Khalid, The Unconventional Replacement Algorithms, Computer Architecture News, Vol. 23, No. 5, December 1995, pp. 20–26

[11] C. Lund, S. Phillips and N. Reingold. IP-paging and distributional paging. In *Proceedings of 35rd Annual Symposium on Foundations of Computer Science*, 1994.

[12] E.J. O'Neil, P.E. O'Neil, and G. Weikum, The LRU-K Page Replacement Algorithm For DataBase Disk Buffering, SIGMOD Record, Vol. 22, Issue 2, June 1993, pp. 297–306.

[13] Przybylski, S.A., Cache Design: A Performance-Directed Approach, Morgan Kaufmann Publishers, 1990.

[14] S. Phillips, Personal communication.

[15] P. Scheuermann, J. Shim, and R. Vingralek, WATCHMAN: A Data Warehouse Intelligent Cache Manager, Proceedings of the 22nd VLDB Conference Mumbai (Bombay), India, 1996.

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

[17] N. Young, On-Line Caching as Cache Size Varies, Proceedings of the 2nd annual ACM-SIAM Symposium Discrete Algorithms, 1991, pp. 241–250.