Contents

Chapter 1

Introduction

1.1 Introduction

The paging problem has inspired several decades of theoretical and applied research and became a classical problem of computer science. It is due to the fact that managing a two level storage system has been and continues to be a task of great practical importance. Although the formulation of the problem is in terms from operating systems like pages and frames, it has applications in other areas, such as assigning communication channels to IP connections in connection-oriented networks or caching web pages.

In the thesis, we consider only the classical paging problem, i.e. we assume that the size of each page as well as its associated cost is unitary. We study only deterministic paging algorithms, although some of the algorithms introduced in Chapter 3 can be randomized. The thesis consists of two major parts.

In the first part (Chapter 2) we study the behavior of the optimal offline paging algorithm. We devise an "online" criterion that allows us to efficiently determine when does the optimal algorithm fault and, using this criterion, we show how to efficiently implement the optimal algorithm. We also observe the behavior of the optimal algorithm on various graphs in the access graph model and we identify structures in the graph that make the optimal algorithm behave badly.

In the second part (Chapter 3) we propose two possible restrictions on the request sequences that can, as we believe, model the locality of reference observed in the request patterns. We give a strongly competitive algorithm for one of these models and an algorithm that is strongly competitive in a subset of the model.

Each section begins with an overview of the previous work relevant to the topic. Concepts and notation used throughout the thesis are introduced in the next section.

1.2 Basic notation and definitions

The general problem deals with a two-level store consisting of a small fast memory (the *cache*) consisting of k *frames* (slots) and a slow memory that holds n ($n > k$) objects called *pages*.

A paging algorithm is presented with a sequence of requests to the pages. It must service each request by retrieving the requested page from the cache. If the requested page p is in the cache (a *hit*), the algorithm incurs no cost. If not (a *fault*), the page must be brought into cache at a unit cost. A "good" paging algorithm tries to minimize the cost of processing a page request sequence, i.e. the number of faults. The cache can hold at most k pages at a time. Therefore, it is sometimes necessary to evict a page from the cache in order to make room for the newly requested page. Without loss of generality we shall assume that evictions are done only at faults, when all k frames are occupied.

A paging algorithm is called *online*, if it bases its decisions on past requests and the contents of the cache only, without knowing future requests. (Almost) all algorithms of practical interest are online, including the Least Recently Used (LRU) and First In First Out (FIFO) algorithms. Algorithms that know the whole sequence in advance are called *offline*.

We now introduce some notation used in the thesis. For request sequences we usually use the symbol σ, the particular requests are denoted by s_i: $\sigma = s_1, s_2, \ldots, s_l$. We may omit commas in the representation of σ. We use $|\sigma|$ for the length of the request sequence. We view request sequences as words over an n letter alphabet, which allows us to use standard notation from the theory of formal languages for operations on the sequences like $\cdot$ for the concatenation of two sequences or * for the iteration of a sequence. If it will not lead to misunderstandings, we shall identify the set of pages P with the set $\{1, \ldots, n\}$ of the first n positive integers.

We denote by C_t the set of pages residing in the cache at time t. For every initial cache contents C_0 and a request sequence $\sigma = s_1 \ldots s_l$, the paging algorithm A defines the sequence $C_1, C_2, \ldots, C_l$ with the following properties: (i) $C_t = C_{t-1}$, if $s_t \in C_{t-1}$, (ii) $C_t = C_{t-1} \cup \{s_t\}$ if $s_t \notin C_{t-1}$ and $|C_{t-1}| < k$, (iii) $C_t = C_{t-1} \cup \{s_t\} \setminus \{e_t\}$ if $s_i \notin C_{t-1}$ and $|C_{t-1}| = k$, where e_t is the page evicted at time t. For an online algorithm, C_t depends only on $s_1 s_2 \ldots s_t$ and C_{t-1}; for an offline algorithm, C_t may depend on the entire σ. The *cost* of A on σ is the number of such t ($1 \le t \le l$) that $s_t \notin C_{t-1}$ and it is denoted by $A(\sigma)$. In what follows we shall say that s_t is a hit if $s_t \in C_{t-1}$ and s_t is a fault if $s_t \notin C_{t-1}$. No confusion should arise from considering s_t to be a page, a request to a page, and a hit/fault.

We shall use the notation $C_t^{A(\sigma)}$ to stress that the cache is a result of an algorithm A operating on the first t requests of σ. We may omit σ or A if it is clear from the context. Let us denote by $e_i^{A(\sigma)}$ the page evicted at the request s_i of σ (let $e_i^{A(\sigma)}$ be the empty word ε if s_i is a hit for A). Note that for $\sigma = s_1 s_2 \ldots s_l$

$A(\sigma) = |e_1^{A(\sigma)} e_2^{A(\sigma)} \dots e_l^{A(\sigma)}|$.

We often do not wish the request sequence σ to be arbitrary, but we want it to satisfy some constraint.

The most widely studied constraint is the following: for a given graph G on n vertices, we require σ to be a walk (a sequence of vertices connected by edges) on G. This model was first introduced in [10] and we shall discuss it in sections 2.4 and 3.1. Other types of constraints will be introduced in Chapter 3. In general, we may wish σ to belong to some class Σ. For a given class Σ, we may study the behavior of an algorithm on request sequences belonging to this class, or we may wish to find the best algorithm, according to some measure, for the given class.

The *cost to length ratio* of an algorithm A on a request sequence σ (with the initial cache contents C_0) is the number of faults incurred by A on σ divided by the length of σ. The *fault ratio* of A on a request sequence class Σ is the infimum of all numbers c, such that there is a constant b such that for every $\sigma \in \Sigma$,

$$A(\sigma) \leq lc + b$$

where $l = |\sigma|$. We denote it by $f_A(\Sigma)$.

If σ can be arbitrary, any online algorithm can be forced to fault on every request by always requesting a page outside its fast memory. Thus, the fault ratio is not a good measure for online algorithms. Early works in paging tried to analyze online algorithms assuming a fixed probability distribution of the requests. However, it is very difficult to propose a suitable probabilistic model, since the request pattern changes dramatically with time. A better measure of performance of online algorithms is the *competitive ratio*, proposed by Sleator and Tarjan [3]. We compare the cost of the online algorithm A to the cost of the optimal offline algorithm OPT. We say that an algorithm A is c-competitive for a request sequence class Σ, if there is a constant b such that

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

for all $\sigma \in \Sigma$. The competitive ratio of A on Σ is an infimum of all such numbers for which A is c-competitive for Σ. It is denoted by $c_A(\Sigma)$. The (online) competitive ratio $c(\Sigma)$ of a class Σ is infimum of $c_A(\Sigma)$ over all online algorithms A.

The class of allowed request sequences $\Sigma = \Sigma(\Pi)$ is often parametrized by some parameter Π. For example in the access graph model, there is a class $\Sigma(G)$ of request sequences for every graph G. A parametrized class $M : \Pi \to \Sigma(\Pi)$ of request sequences we call a *model*. We say that an algorithm A is strongly competitive for a model M if $c_A(\Sigma(\Pi)) = O(c(\Sigma(\Pi)))$, i.e. if there are constants a and b such that $c_A(\Sigma(\Pi)) \leq a\,c(\Sigma(\Pi)) + b$ for all values of the parameter Π from the respective domain.

Chapter 2

The optimal offline algorithm

2.1 The optimal algorithm

The optimal offline algorithm for paging is the Belady's algorithm [2]. This algorithm, called OPT, incurs on every request sequence at most as many faults as any other paging strategy. OPT on a fault always evicts a page whose next request is furthest in the future among all pages in the cache.

Lemma 2.1 *Let $\sigma = s_1 s_2 \dots s_l$ be a request sequence. On any prefix $s_1 \dots s_i$ of σ, OPT incurs at most as may faults as any other paging algorithm for servicing σ. (We assume that both algorithms start with the same cache contents.)*

Proof. Let A be a paging algorithm. We shall prove that for each i, $\mathrm{OPT}(s_1 \dots s_i) \leq A(s_1 \dots s_i)$ by constructing inductively a sequence of algorithms $A_0, A_1, \dots A_l$ such that for each $i \geq 0$

$$A_i(s_1 \dots s_j) = \mathrm{OPT}(s_1 \dots s_j) \text{ and } A_i(s_1 \dots s_j) \leq A(s_1 \dots s_j) \text{ for all } j \leq i \quad (*).$$

Let $A_0 = A$. Clearly, on empty sequence (*) holds. Let us construct A_{i+1} from A_i. Let $e_{i+1}^{A_i} = e_{i+1}^{\mathrm{OPT}}$. Then for $A_{i+1} = A_i$ the (*) holds. Suppose $e_{i+1}^{A_i} \neq e_{i+1}^{\mathrm{OPT}}$. Let us denote $e_{i+1}^{A_i} = p$ and $e_{i+1}^{\mathrm{OPT}} = q$. We shall construct A_{i+1} from A_i as follows.

A_{i+1} evicts q and keeps p in the cache until A_i evicts q or until q is requested. In the first case, A_{i+1} simply evicts p. In the second case, p must be requested before q. Since p is not in the cache of A_i, A_i faults but A_{i+1} does not. From this point on, the cache contents of A_i and A_{i+1} differ in at most one page. If A_{i+1} faults, but A_i does not, A_{i+1} synchronizes the caches by evicting the page that is not in cache of A_i. If both algorithms fault, A_{i+1} evicts the page evicted by A_i (if the page is in A_{i+1}'s cache), or the page that is not in cache of A_i. After synchronization, A_{i+1} always evicts the same page as A_i.

It thus follows that for each i $\mathrm{OPT}(s_1 \dots s_i) \leq A(s_1 \dots s_i)$. □

Lemma 2.2 *Let* $\sigma = s_1 s_2 \dots s_l$ *and* $\sigma' = s'_1 s'_2 \dots s'_{l'}$ *be request sequences and let for some* h $s_1 s_2 \dots s_h = s'_1 s'_2 \dots s'_h$. *Then* $|e_1^{\mathrm{OPT}(\sigma)} e_2^{\mathrm{OPT}(\sigma)} \dots e_h^{\mathrm{OPT}(\sigma)}| = |e_1^{\mathrm{OPT}(\sigma')} e_2^{\mathrm{OPT}(\sigma')} \dots e_h^{\mathrm{OPT}(\sigma')}|$, *i.e. the number of faults incurred by OPT while servicing the prefix* $s_1 \dots s_h$ *does not depend on the rest of the sequence.*

Proof. Let C_t^{σ} ($C_t^{\sigma'}$) be the cache of OPT servicing σ (σ') after the t-th request. $C_0^{\sigma} = C_0^{\sigma'}$ and by induction we can easily show that for all pages p such that $p = s_k$ for some $t < k \leq h$, p is in C_t^{σ} if and only if it is in $C_t^{\sigma'}$. It means that OPT incurs fault on σ at time t if and only if it incurs a fault on σ' at time t. □

Lemma 2.3 *Consider* n *pages* $\{1, 2, \dots n\}$ *and a cache of size* k. *OPT never incurs more than* $\left\lceil \frac{l}{n-1} \right\rceil (n-k)$ *faults on a sequence of length* l.

Thus, the fault ratio of OPT is $\frac{n-k}{n-1}$.

Proof. (adopted from Sleator and Tarjan, [3]) We divide the request sequence into phases of length $n-1$. Let C be the cache contents at the beginning of a phase. In this phase, there must be at least $k-1$ requests to pages either initially contained in the cache or requested earlier in the phase. We can adopt the following strategy: we preserve the first $k-1$ pages that will be requested further in the phase in the cache until the end of the phase. This way we achieve at least $k-1$ hits in the phase. OPT incurs at most as many faults as our strategy, so the statement holds. □

Since the initial cache contents C can be arbitrary, we have also shown the following.

Lemma 2.4 *OPT never incurs more than* $n-k$ *faults servicing any subsequence of length* $n-1$.

Moreover, this bound is tight. Consider a request sequence of the form $(p_1 p_2 \dots p_{n-1} p_n p_{n-1} \dots p_2)^*$. Each phase of length $n-1$ consists of $n-1$ different pages and therefore OPT must fault at least $n-k$ times in each phase. Another example is a sequence of the form $(p_1 p_2 \dots p_n)^*$.

Lemma 2.5 (Fiat et al., [8]) *On any request sequence* σ, *the inequality* $\mathrm{OPT\_MARK}(\sigma) \leq 2\ \mathrm{OPT}(\sigma)$ *holds. We assume that both algorithms start with the same cache contents.*

Proof. We divide the request sequence into phases. The first phase is the longest prefix of the request sequence, that contains requests to no more than k different pages. Similarly, each next phase is the longest segment of the request sequence containing requests to no more than k different pages and starting at the first request not belonging to the previous phase. Each marking algorithm has at the end of the phase all pages in the cache marked. The first page of the next phase is by definition outside its cache, thus it must cause a fault. At this fault, all pages (except the newly requested one) become unmarked.

For the i-th phase, let g_i be the number of pages, that are requested in this phase but were not requested in the $(i-1)$-th phase (or were outside the cache at the beginning, if $i = 1$). Clearly, OPT_MARK does not fault more than g_i times in the i-th phase (it faults exactly g_i times).

In the (i-1)-st and i-th phase, $k + g_i$ different pages are requested. Since OPT has only k pages in the cache, it must incur at least g_i faults in the two phases. By summing all pairs of adjacent phases, we have $\mathrm{OPT}(\sigma) \geq \sum_i g_{2i}$ and also $\mathrm{OPT}(\sigma) \geq \sum_i g_{2i+1}$. By adding these two inequalities we obtain $2\,\mathrm{OPT}(\sigma) \geq \sum_i g_i = \mathrm{OPT\_MARK}(\sigma)$. □

This bound is also tight. If we denote $\sigma_1 = p_1 p_2 \dots p_k$ $\sigma_2 = p_{k+1} \dots p_{2k}$, $\sigma_1^R = p_k p_{k-1} \dots p_1$ $\sigma_2^R = p_{2k} p_{2k-1} \dots p_{k+1}$, then on the repeated sequence $(\sigma_1 \sigma_2 \sigma_1^R \sigma_2^R)^*$ OPT_MARK faults on every request, but OPT incurs only k faults in every two phases.

2.2 OPT fault criterion

OPT is an offline algorithm. It means, that it needs to know the request sequence in advance to determine which page to evict. But, as we have seen in Lemma 2.2, the number of faults incurred by OPT on a prefix of the request sequence does not depend on the rest of the sequence. Let $\mathit{fault}(t) = \mathit{fault}(C_0, s_1 \dots s_{t-1}, s_t)$ be a predicate that is true if and only if OPT faults at time t at the request s_t. We may omit parameters C_0 and $s_1 \dots s_{t-1}$ if known from the context. Since $\mathit{fault}(t)$ is true if and only if the cost paid by OPT increases by 1 at time t (i.e. $\mathrm{OPT}(s_1 \dots s_t) = \mathrm{OPT}(s_1 \dots s_{t-1}) + 1$), the predicate depends only on the initial cache contents C_0 and the request sequence until time t. The value of the predicate can obviously be determined by running OPT on $s_1 \dots s_t$. A drawback of this approach is that to determine a single value of the predicate, we need to know the entire history of requests, which may be potentially very long.

Our aim is to represent all the information needed to determine $\mathit{fault}(t)$ in a compact form, i.e. to have $\mathit{fault}(t) = F(\phi_{t-1}, s_t)$, where ϕ_{t-1} is a (compact) state information extracted from C_0 and $s_1 \dots s_{t-1}$. We also want to efficiently compute the new state ϕ_t given the old state ϕ_{t-1} and the request s_t. We show that ϕ_t can be stored using $O(n)$ integers (or $O(n \log n)$ bits) and that we can compute ϕ_t from ϕ_{t-1} in time $O(\log n)$. In the same time we are able to evaluate $F(\phi_{t-1}, s_t)$. This allows us to construct an optimal offline paging algorithm whose running time is $O(l \log n)$, where l is the length of the request sequence.

Let $f_{p,t}$ be the number of faults incurred by OPT since p was last time requested until the time t. If the last request to p is a fault, it does not count in

$f_{p,t}$, i.e. if the request $s_t = p$ is a fault, $f_{p,t} = 0$. Initially, we define $f_{p,0}$ to be 0 if p is in the cache at time 0 and $n - k$ if it is not (we assume that there are no empty frames in the cache). We write f_p, if the time t is known from the context. We use the notation $\overline{f} = (f_1, \ldots, f_n)$ and $\overline{f}_t = (f_{1,t}, \ldots, f_{n,t})$. We shall call $\overline{f}$ the *fault vector*.

First observe the way the fault vector $\overline{f}$ changes. Let $p = s_t$ be the request at time t. If we are given $\overline{f}_{t-1}$, we can compute $\overline{f}_t$ using one of the two following update rules:

1. If not *fault*(t), then $f_{q,t} = f_{q,t-1}$ for all pages $q \neq p$ and $f_{p,t} = 0$.
2. If *fault*(t), then $f_{q,t} = f_{q,t-1} + 1$ for $q \neq p$ and $f_{p,t} = 0$.

While it is possible to have f_p arbitrarily large for some p, there cannot be many pages having "small" f_p.

Fault Criterion. *Fault vector $\overline{f} = (f_1, \ldots, f_n)$ satisfies the fault criterion, if for every i, $i \geq 0$, there are at most $i + k$ pages p, such that $f_p \leq i$.*

Let $\overline{f}_0, \overline{f}_1, \ldots, \overline{f}_l$ be a sequence of fault vectors corresponding to a run of a paging algorithm A on $\sigma = s_1 \ldots s_l$.

Lemma 2.6 *At every time t, the fault vector $\overline{f}_t$ satisfies the fault criterion.*

Proof. If $i \geq n - k$, the claim is obvious, since the total number of pages is n. For a given $i < n - k$, let S, $|S| \geq k$ be a set of pages such that $f_p \leq i$ for all $p \in S$. Let q be the least recently requested page from S at time t (for any page p that was never requested, $f_p \geq n - k$, therefore every page in S was requested before t) After q, $|S| - 1$ different pages have been requested. Since at most $k - 1$ out of these pages could be in the cache immediately after the request to q, A incurred at least $|S| - 1 - (k - 1)$ faults since that request. Therefore $|S| - k \leq f_q \leq i$, thus $|S| \leq i + k$. $\square$

Let $\sigma = s_0 s_1 \ldots s_l$ be a request sequence. We use the fault criterion to formally construct a sequence $\overline{f}_0, \overline{f}_1, \ldots, \overline{f}_l$ of fault vectors for the given σ. Each $\overline{f}_t$ satisfies the criterion and we always use the first update rule to compute $\overline{f}_t$ from $\overline{f}_{t-1}$, unless it contradicts the fault criterion. In the latter case, we use the second rule. Let *ffault*(t) be a predicate that is true if and only if $\overline{f}_t$ was constructed using the second rule. Obviously, *ffault*(t) depends only on $\overline{f}_{t-1}$ and s_t. If *ffault*(t) holds, we say that s_t is a *formal fault*, otherwise s_t is a *formal hit*.

We thus declare a particular s_t to be a (formal) hit or fault solely based on the applicability of the update rules for fault vectors. At this point, we cannot assume the existence of a paging algorithm whose way of organizing cache would result in such a fault vector sequence. Later on, we present an algorithm OPT2 that services the request sequence and faults at time t $(1 \leq t \leq l)$ if and only if *ffault*(t) is true. Then we show that OPT faults at time t if and only if OPT2 faults.

The next lemma applies to the formally constructed sequence $\overline{f}_0, \overline{f}_1, \ldots, \overline{f}_l$ above.

Lemma 2.7 *At each time t, there is an i ($0 \le i \le n-k$) such that there are exactly $i+k$ pages p such that $f_p \le i$. Let i_t denote the least such i at time t. Then, $i_{t+1} = i_t + 1$ for a formal fault at $t+1$ and $i_{t+1} \le i_t$ for a formal hit.*

We say that i_t is the *finger* of the vector $\overline{f}_t$.

Proof. The proof is by induction on t. For $t = 0$ the finger is $i_0 = 0$. Assume $t \ge 0$ and let $S_{j,t}$ be the set of pages such that $f_{p,t} \le j$.

Let $s_t = p$ be the $(t+1)$-st request to a page such that $f_{p,t} > i_t$. If it was a formal hit, then $S_{i_t,t+1} = S_{i_t,t} \cup \{p\}$. By induction hypotesis $|S_{i_t,t}| = i_t + k$. But then $|S_{i_t,t+1}| = |S_{i_t,t}| + 1 > i_t + k$ which violates the fault criterion at time $t+1$. Therefore the $(t+1)$-st request is a formal fault. We thus have $S_{j,t+1} = S_{j-1,t} \cup \{p\}$ for all $0 \le j \le f_{p,t}$ (we define $S_{-1,t} = \emptyset$). Since for all $j < i_t$, $|S_{j,t}| < j + k$ and $|S_{i_t,t}| = i_t + k$, we have also $|S_{j,t+1}| = |S_{j-1,t}| + 1 < j - 1 + k + 1$ for $j \le i_t$ and $|S_{i_t+1,t+1}| = i_t + k + 1$ which gives $i_{t+1} = i_t + 1$.

On the other hand, let the $(t+1)$-st requested page $s_t = p$ be such that $f_{p,t} \le i_t$. We show that if we compute $\overline{f}_{t+1}$ using the first rule, the fault criterion remains satisfied at time $t+1$ and hence, the request is a formal hit. Let $\overline{f}_{t+1}$ be computed using the first rule. Then, the size of sets S_j for $j < f_{p,t} \le i_t$ increases by one (namely by the element p) the sets S_j for $j \ge f_{p,t}$ remain unchanged. Since $|S_{j,t}| \le j + k + 1$, $|S_{j,t+1}| \le j + k$ for $j < i_t$, the fault criterion remains satisfied also at time $t+1$. Since $S_{i_t,t} = S_{i_t,t+1}$, $|S_{i_t,t+1}| = i_t + k$ and thus $i_{t+1} \le i_t$. □

Proposition 2.8 *The request $s_{t+1} = p$ is a formal fault if and only if $f_{t,p} > i_t$.*

Proof. This fact has already been proven in the proof of Lemma 2.7. □

2.2.1 The OPT2 algorithm

Now we are ready to describe the algorithm OPT2. Let $\sigma = s_1 \ldots s_l$ be a request sequence and let $\overline{f}_1, \ldots \overline{f}_l$ be a corresponding sequence of fault vectors. OPT2 will service σ and will fault at time t if and only if *ffault*(t) is true.

The usual approach in the design of paging algorithms is to decide at each fault which page to evict. We employ a different idea. Let the page $p = s_t$ be requested at time t and let s_t be a formal fault. We thus know that p is not in the cache at time t (we are designing the algorithm to behave this way). Instead of deciding which page to evict at time t, we find at which fault the page p should have been evicted from the cache.

There are two cases. If p was neither in the cache at time 0 nor has been requested until t, p has never been in the cache until t and therefore could not have been ever evicted. We work around this by assuming that p was evicted at a "virtual fault" in some time $t' \leq 0$. Since there are $n - k$ pages outside the cache at time 0, there must be $n - k$ such "virtual faults". We define all the virtual faults to happen simultaneously at time 0. More precisely, we define $n-k$ "virtual times" $[0, 1]$, $[0, 2]$, $\ldots$, $[0, n - k]$ and we define $\mathit{ffault}([0, j])$ $(1 \leq j \leq n - k)$ to be true.

For the second case, let the most recent access to p be at time $t' < t$. Then p might be evicted at any fault during the interval $\langle t' + 1, t - 1\rangle$. In other words, it could be evicted at any of the $f_{p,t-1}$ least recent faults.

The output of the paging algorithm is the sequence $e_1, e_2, \ldots e_l$, where e_t is the page evicted at time t (if $\mathit{ffault}(t)$ is true) or $e_t = \varepsilon$ (if $\mathit{ffault}(t)$ is false). The output is valid if

- at each time t, such that $\mathit{ffault}(t)$ is true, exactly one page is evicted
- e_t is in cache at time $t - 1$
- the first request to page e_t after time t, if such a request exists (let it be at time $t' > t$) is a fault (i.e. $\mathit{ffault}(t')$ holds)
- the first request (let it be at time t') to a page that is not in the cache at time 0 is a fault (i.e. $\mathit{ffault}(t')$ holds).

Our algorithm will proceed as follows. First we set all e_t to be ε. We add $n-k$ "virtual elements" $e_{[0,1]}, e_{[0,2]}, \ldots e_{[0,n-k]}$ and also set them to ε. These correspond to the virtual faults. Recall that for each page p, $f_{p,0} = 0$ if $p \in C_0$ and $f_{p,0} = n-k$ if $p \notin C_0$. Observe that since $i_0 = 0$, the first request to each page $p \notin C_0$ will be a formal fault.

We will process the request sequence σ sequentially and stop at each formal fault. Let the request $s_t = p$ be the first unprocessed formal fault. For this fault, we find minimal $t^- < t$ such that

- $\mathit{ffault}(t^-)$ holds
- $e_{t^-} = \varepsilon$
- for at most $f_{p,t-1}$ times t' such that $t^- \leq t' < t$, $\mathit{ffault}(t')$ holds

We allow the case $t^- = [0, i]$. We define $[0, i] < j$ for all $i, j > 0$. The elements $[0, i]$ are incomparable (we may choose any of them). We say that the fault at time t *covers* the fault at time t^-. We set $e_{t^-} = p$ and proceed to the next formal fault until no unprocessed faults remain. Since for the least recent formal fault (at time $t' < t$), the corresponding $e_{t'}$ is always ε, and every request to a page for which $f_p = 0$ is a formal hit, the t^- always exists.

Observe that if $p \in C_0$ or p has been requested before time t, $f_{p,t-1}$ is at most equal to the total number of formal faults (not including virtual faults) until time $t - 1$. Hence, $t^- = [0, j]$ for some j if and only if $p \notin C_0$ and p is for the first time requested at time t. Thus, virtual faults are covered only by first requests to pages that were outside the cache at time 0.

After processing all faults, $n-k$ uncovered faults (including uncovered virtual faults) remain. At each such fault, we must evict a page. The only possible candidate is a page, that is not requested after the fault until the end of the request sequence. We show that there are enough pages that have not been requested "long enough" in order to cover each fault that remains uncoverered at time l.

Notation: Let $\langle t_j^-, t\rangle$ be the smallest closed time interval that ends at time t containing at least j, $j \geq 0$ formal faults. If $t_j^- = [0, i]$ for some i, we make an agreement that the interval contains also all virtual faults. Let $D_{j,t}$ be the set of all uncovered faults in this interval at time t (i.e. $D_{j,t}$ contains all t', such that neither $t_j^- > t'$ nor $t' < t$ and that $e_{t'} = \varepsilon$ and $\mathit{ffault}(t')$ holds).

Assume for a moment that the following proposition holds. Recall that $S_{j,t}$ is the set of pages p, for which $f_{p,t} \leq j$.

Proposition 2.9 *At every time t the following holds:*

a) for every $0 \leq j \leq i_t$, $|D_{j,t}| \geq j$

b) for every $j \geq 0$, $|D_{j,t}| + k \geq |S_{j,t}|$.

Part b) of the proposition at time l implies that for each uncovered fault s_t there is a matching page p such that p is not requested after time t (nor $s_t = p$). In fact, this is an example of a bipartite matching, where page p can be matched to a fault s_t if and only if $t \in D_{f_{p,l},l}$. Let S be a set of pages. Let $N(S)$ be a set of neighbors of pages in S, i.e. $N(S)$ contains all uncovered faults that can be matched to at least one page in S.

Theorem 2.10 (marriage theorem) *Let for any set $S \subset P$ be $|N(S)| \geq |S| - k$. Then all but k elements of P can be matched.*

Proof. Proof of this theorem can be found in most textbooks on graph theory, e.g. in [13].

Take as P the set of all n pages. Let $f = \max(\{f_{p,l} \mid p \in S\})$. Then $N(S) = D_{f,l}$, $S \subset S_{f,l}$, hence $|N(S)| = |D_{f,l}| \geq |S_{f,l}| - k \geq |S| - k$, and by the marriage theorem, $n-k$ pages can be matched. Thus all $n-k$ uncovered faults can be matched.

Proof of proposition 2.9. The proof is by induction on t. For $t = 0$, $i_0 = 0$ and the statement holds. Assume the proposition holds at time $t-1$.

Let us first assume that the request $s_t = p$ is a formal hit. Then, $D_{j,t} = D_{j,t-1}$ for all $j \geq 0$. Since $i_t \leq i_{t-1}$, part a) clearly holds. For part b), observe that $S_{j,t} = S_{j,t-1}$ for $j \geq f_{p,t-1}$. According to Proposition 2.8, $f_{p,t-1} \leq i_{t-1}$, which means that the sets $S_{j,t} = S_{j,t-1}$ remain unchanged for $j \geq f_{p,t-1}$. For $j < i_{t-1}$ the claim is implied by the part a) of the induction hypotesis and the fact that the fault criterion $|S_j| \leq j + k$ is satisfied.

Now assume that the request $s_t = p$ is a formal fault. Let the fault covered by s_t be the request $s_{t'}$. Let f' be the number of (non-virtual) faults in the interval $\langle t', t-1\rangle$. Let $f = f_{p,t-1}$.

$$\begin{array}{rcll} S_{j,t} & = & S_{j-1,t-1} \cup \{p\} & \text{for } j \le f \\ S_{j,t} & = & S_{j-1,t-1} & \text{for } j > f \\ D_{j,t} & = & D_{j-1,t-1} \cup \{t\} & \text{for } j \le f' \\ D_{j,t} & = & D_{j-1,t-1} \cup \{t\} \setminus \{t'\} & \text{for } j > f' \end{array}$$

From Proposition 2.8 we have $f_{p,t-1} = f > i_{t-1}$. Thus For $j > f$, part b) holds at time t.

Let $f' < j \le f$. Observe that $D_{f,t-1} = D_{f-1,t-1} = \ldots = D_{f',t-1}$, $|D_{f',t-1}| > |D_{f'-1,t-1}|$ and $|D_{f,t-1}| \ge |S_{f,t-1}| > |S_{f-1,t-1}| \ge \ldots \ge |S_{f',t-1}|$. This implies $|D_{f-1,t-1}| + k \ge |S_{f-1,t-1}| + 1$, $|D_{f-2,t-1}| + k \ge |S_{f-2,t-1}| + 1 \ldots |D_{f',t-1}| + k \ge |S_{f',t-1}| + 1$. Since $D_{j-1,t-1}$ is at least by one greater than $S_{j-1,t-1}$, part b) of the proposition holds also for $f' < j \le f$. For $j \le f'$ part b) obviously holds.

It remains to prove part a). If $f' > i_{t-1}$, the claim directly follows from the induction hypotesis.

Let $f' \le i_{t-1}$. Observe that $D_{f,t-1} \ge |S_{f,t-1}| > |S_{i_{t-1},t-1}| = i_{t-1}$. The last inequality is true because $p \in S_{f,t-1}$ but $p \notin S_{i_{t-1},t-1}$. Together with the fact that $D_{f',t-1} = D_{f'+1,t-1} = \ldots = D_{f,t-1}$, we get the desired claim. □

Theorem 2.11 *Let σ be a request sequence and let $\overline{f}_0, \ldots, \overline{f}_l$ be a corresponding sequence of formal fault vectors. OPT faults on σ at time t if and only if ffault(t) holds.*

Proof. By contradiction, let t be the first time such that OPT either (1) hits on a formal fault or (2) faults on a formal hit. Consider the first case. Since until t OPT faulted exactly on formal faults, $\overline{f}_{t-1}$ is a valid fault vector for OPT. If OPT would hit, its fault vector would not satisfy the fault criterion, what contradicts Lemma 2.6. The second case can never happen, because OPT incurs at most as many faults on the prefix $s_1 s_2 \ldots s_t$ as any other algorithm, in particular as many as OPT2. □

Theorem 2.12 *Let σ be a request sequence, let $\overline{f} = \overline{f}_t$ be the fault vector of OPT servicing σ at time t and let i_t be the finger of $\overline{f}_t$. Then OPT faults on the next request $p = s_{t+1}$ if and only if $f_{p,t} > i_t$.*

Proof. Since OPT faults exactly at formal faults, its fault vector is equal to the formal fault vector. By the Proposition 2.8, formal fault at time $t+1$ occurs if and only if $f_{s_{t+1},t} > i_t$. □

Theorem 2.13 *The size of the state information ϕ_t needed to compute the predicate $fault(t+1) = F(\phi_t, s_{t+1})$ is finite and ϕ_t can be stored using $O(n \log n)$ bits.*

Proof. Since $fault(t+1) \iff ffault(t+1)$, we can discuss the minimum information needed to compute $ffault(t+1)$. We cannot take $\overline{f}_t$ for ϕ_t, since the values $f_{p,t}$ can be unbounded for some pages p. Since OPT faults on s_{t+1} if and only if $f_{s_{t+1}} > i_t$, we do not need to store the exact values $f_{p,t} > i_t$. For these pages we store only the substitute value ∞ (infinity). Since $0 \le i_t \le n-k$, for each f_p we need only $\lceil \log_2(n-k+2) \rceil$ bits. $\square$

The theorem can be rephrased as follows: there are no more than $(n-k+2)^n$ possible different state vectors ϕ_t.

2.3 Efficient implementation of OPT2

For the purposes of a simulation, it might be useful to implement the algorithm OPT on a computer. A straightforward implementation is as follows. For each page p we maintain a boolean value $incache(p)$ and an integer $next(p)$ which is the time of the next request to the page. Each time a page is requested, we recompute $next(p)$ for this page. If $incache(p)$ is true, we proceed to the next request. If not, we find a page q for which $incache(q)$ is true and has the largest $next(p)$ attribute. We evict the page q by setting $incache(q)$ to false, $incache(p)$ to true and continue with the next request. In this implementation, the running time is $\Theta(nl)$, where n is the number of pages and l is the length of the sequence. This is easy to see, because for each page p, the accumulated cost of updating $next(p)$ is $\Theta(l)$.

We now show how to efficiently implement the algorithm OPT2 that faults if and only if OPT faults to achieve running time $O(l \log n)$ on a request sequence of length l. Note that OPT2 is not necessarily identical to OPT, since on a fault it may choose to evict a different page than OPT. We briefly sketch the idea.

The main point is to maintain the fault vector $\overline{f}$ in a way that allows fast (in time $O(\log n)$) updates. A suitable data structure for this is some kind of a balanced binary search tree, such as the Red-Black tree. For a description of the Red-Black trees refer for example to [12]. We augment the data structure with additional information stored in the internal nodes that is needed to efficiently verify the fault criterion. More exactly, we need the tree structure to quickly compute the finger i_t of $\overline{f}$ defined in Theorem 2.12.

For each page there is a leaf of the tree. It holds the pair (p, f_p). We shall use the same symbol (e.g. p) for a page and its corresponding leaf in the tree. Each page keeps a pointer to its leaf. Every internal node has two children. We assume that each internal node v has pointers $left(v)$ and $right(v)$ to its left and right child and (except for the root) a pointer $up(v)$ to its parent. The search key is

f_p, i.e. for each internal node the following property holds: for every page p from the subtree rooted at the left child and every page q from the subtree rooted at the right child, $f_p \leq f_q$.

At a fault, we need to increment each f_p by 1. We can not afford $\Theta(n)$ operations to update each f_p directly. We use an indirect scheme instead. For each page we store a value g_p instead of f_p and keep a global offset o, such that for every p, $f_p = g_p + o$. Thus, we need only to increment the offset o. This mechanism shall be transparent in further discussion, e.g. each time we refer to f_p we mean the value $g_p + o$, to set f_p to a value of x means to assign the value $x - o$ to g_p. Note that incrementing all f_p does not change their relative order and thus does not violate the search tree property.

Let $p_1, p_2, \ldots p_n$ be the leaves of the tree listed from left to right (i.e. $f_{p_1} \leq f_{p_2} \leq \cdots \leq f_{p_n}$).

- Let i be such that $f_{p_{i+k}} - i$ is minimal (minimum is taken over all $0 \leq i \leq n-k$).
- Let i_f be the smallest number with this property.

From the Lemma 2.6 we have $|S_{i-1}| \leq i + k - 1$, or equivalently $p_{i+k} \notin S_{i-1}$, and thus $f_{p_{i+k}} > i - 1$. Hence, $f_{p_{i+k}} - i \geq 0$ for all $0 \leq i \leq n-k$. For each i such that $|S_i| = i + k$ we have $f_{p_{i+k}} \leq i$ which means that the minimum $f_{p_{i+k}} - i = 0$ is achieved. Conversely, if $f_{p_{i+k}} \leq i$, then $|S_i| \geq i + k$. Thus, i_f is really the finger of $\overline{f}$. Observe that since $f_{p_{i+k}} - i > 0$ for $i < 0$, we may take the minimum over all i such that $-k < i \leq n - k$.

For each internal node v we store three additional attributes $min(v)$, $size(v)$ and $mindif(v)$, where $min(v)$ is the minimum value of f_p of the leaves in the subtree rooted at v, $size(v)$ is the number of leaves of the subtree. (Note that for the min attribute we use the same offset technique as for f_p).

Let $p_1, p_2, \ldots p_m$ be the leaves of the subtree rooted at v, listed from the leftmost to the rightmost leaf (note that $min(v) = f_{p_1}$). $mindif(v)$ is the value of $\min\{f_{p_i} - f_{p_1} - i | 1 \leq i \leq m\}$

For each leaf p we define

$$\begin{aligned} min(v) &= f_p \\ size(v) &= 1 \\ mindif(v) &= 0 \end{aligned}$$

With this convention, for each internal node v with left and right child v_1 and v_2 we can write

$$\begin{aligned} min(v) &= \min\left(min(v_1), min(v_2)\right) \\ size(v) &= size(v_1) + size(v_2) \\ mindif(v) &= \min\left(mindif(v_1), mindif(v_2) + min(v_2) - min(v_1) - size(v_1)\right) \end{aligned}$$

Since the values of attributes of each node depend only on those of its children, they can be easily updated during insert, delete and rebalancing operations.

The smallest i that minimizes the expression $f_{p_{i+k}} - i$ is computed by a simple recursive function *find_finger*. When called from the root, it returns the current finger i_f.

```
function find_finger(node v)
  if v is a leaf then
    return f_v
  else
    let v_1 and v_2 be the left and right child of v
    if mindif(v_1) = mindif(v) then
      return find_finger(v_1)
    else
      return find_finger(v_2)
```

Let us denote this tree holding the fault vector $\mathcal{F}$. We shall need a second data structure $\mathcal{D}$ that holds a list of all currently uncovered faults, sorted by time when the fault occurred. A balanced search tree is appropriate. For each uncovered fault s, $\mathcal{D}$ shall contain its "distance" d_s, i.e. the number of faults since the request s until the currently processed request s_t.

Our implementation of OPT2 shall work in three phases. In the first phase it builds the search tree for the fault vector $\overline{f}_0$. Since each $f_{p,0}$ contains only one of two possible values (0 or $n-k$), this can be done in time $O(n)$. It builds also the tree $\mathcal{D}$, containing $n-k$ virtual faults.

In the second phase it sequentially processes the request sequence. For each request $s_t = p$

```
let i_f = find_finger(r), where r is the root of F
if f_p > i_f then
  let s_t' ∈ D be such that d_s is the largest such that d_{s_t'} ≤ f_p
  set e'_t = p and delete s_t' from D
  increment all d_s ∈ D and f_p ∈ F
  insert the fault s_t into D
delete p from F
set f_p = 0
insert p into F
```

After processing the last request of the request sequence, the algorithm steps into the third phase. In this phase it covers all uncovered faults. It must match each of the $n-k$ uncovered faults to one of the pages. A fault in distance d can be matched to a page p if and only if $d \leq f_p$. Let $d_1 \leq d_2 \leq \ldots \leq d_{n-k}$ be the distances of uncovered faults after processing the last request and let

$f_{p_1} \leq f_{p_n} \leq \ldots \leq f_{p_n}$ be the elements of the fault vector listed in increasing order. By Proposition 2.9 and Theorem 2.10, the matching exists. We claim that without loss of generality we can match page p_n to the fault in distance d_{n-k} page p_{n-1} to the fault $d_{n-k-1} \ldots$ page k+1 to the fault in distance d_1. This follows easily from the next proposition:

Proposition 2.14 *Take a matching of faults to pages such that the fault s is matched to page q and fault s' is matched to page p. Let $d_s \leq d_{s'}$ be the distances of the two faults and $f_p \leq f_q$. Then it is possible to match s to p and s' to q.*

Proof. Since s' can be matched to p, $d_{s'} \leq f_p$. Thus $d_s \leq d_{s'} \leq f_p \leq f_q$, which implies $d_s \leq f_p$ and $d_{s'} \leq f_q$. □

The matching can be done in time O(n). For each fault s_t that is matched to page p, we set $e_t = p$.

Time and space complexity: the first and third phase run in time $O(n)$, the second phase takes $O(l \log n)$ time, which gives overall time complexity $O(n + l \log n)$. The space required is $O(n)$ of $O(\log(n + l))$ bit integers.

2.4 OPT on access graphs

Although no deterministic online paging algorithm can achieve competitive ratio better than k, where k is the size of the cache (see Lemma 3.1 and Section 3.1), in practical situations many online algorithms behave much better. Another interesting issue is that although both the First In First Out (FIFO) and the Least Recently Used (LRU) algorithms are k-competitive, LRU usually performs better than FIFO in practice. The reason is that the request sequences that occur in practice usually exhibit *locality of reference* – each request is likely to be in a "close neighborhood" of the previously requested page. This led Borodin, Irani, Raghavan and Schieber [10] to propose the *access graph model*. The set of pages is modelled by a graph, in which each vertex corresponds to a page. The locality restriction requires that after a request to a page p, a request to a page q can follow only if $p = q$ or there is an edge from p to q in the access graph. The algorithms we study are *lazy*: they evict a page only at a fault. We may thus assume that after a request to page p, a page $q \neq p$ is requested.

Both directed and undirected access graphs have been studied in the context of online paging algorithms. Although several deep results are known for the undirected case (e.g. [4], [5], [6] and many others), only some special case results are known for the directed case ([10], [9]).

Most of these works are oriented towards design and analysis of good (i.e. competitive) online algorithms for this model. Since the shape of the access graph is determined by the currently running application program and/or data it is processing, we may equally well ask how to write and compile applications (or how to structure data) that would result in "good" access graphs. By "good" we mean a graph for which each request sequence can be serviced with a small number of faults (relative to the length of the sequence).

If the request sequences can be arbitrary, i.e. Σ is the set of all possible sequences of pages $p_1, \ldots, p_n$, the fault ratio of the optimal algorithm is by Lemma 2.1 $f_{\mathrm{OPT}} = \frac{n-k}{n-1}$. We will study the case when Σ is the set of all walks on a given undirected graph G with the set of vertices $P = \{p_1, p_2, \ldots, p_n\}$.

For the given graph G, we would like to determine the fault ratio of the OPT algorithm on the class $\Sigma(G)$ of such request seqences. We write $f_{\mathrm{OPT}}(G)$ as a shorthand for $f_{\mathrm{OPT}}(\Sigma(G))$. We use the term walk for a sequence of vertices $p_1 p_2 \ldots p_l$, such that every two consecutive vertices p_i, p_{i+1} are joined by an edge. The length of the walk is the number of its edges (number of vertices minus 1). A path is a walk that contains every vertex at most once. A covering walk is a walk that contains every vertex; a covering walk that is a path is called a hamiltonian path.

The first theorem in this section deals with the special case when the cache size k is just one less than the total number of pages n.

Theorem 2.15 *Let G, $|G| = n$ be an access graph and let $k = n - 1$ be the size of the cache. Let l be the length of the shortest covering walk on G. Then the fault ratio of OPT is $f_{\mathrm{OPT}}(G) = \frac{1}{l}$. In particular, $f_{\mathrm{OPT}}(G) = \frac{1}{k}$ if and only if G contains a hamiltonian path.*

Proof. Let us first show that $f_{\mathrm{OPT}} \leq \frac{1}{l}$. Assume there was a fault at a request to page p and that page q is evicted. Request to q is the furthest in the future among all pages in the cache. Let us denote σ_{pq} the segment of the request sequence that begins with p and ends just before the first request to q. Page q is requested further in the future than any other page in the cache, therefore σ_{pq} must contain requests to all k pages, that are in the cache after the request to p. The sequence σ_{pq} followed by request to q covers all vertices of G, so the length of σ_{pq} must be at least l. Each request sequence σ, that causes faults at pages $p_1, p_2, \ldots p_i$ can be divided into phases $\sigma_{p_1 p_2}, \sigma_{p_2 p_3}, \ldots \sigma_{p_{i-1} p_i}$ such that page p_{i+1} is evicted at the fault at p_i. Each phase has length at least l, therefore the fault ratio can not be greater than $\frac{1}{l}$.

Next we show $f_{\mathrm{OPT}} \geq \frac{1}{l}$ by constructing an appropriate request sequence. Let $p = p_1, p_2, \ldots, p_{l+1} = q$ be a covering walk of minimum length. Page p_1 occurs in the walk only once (otherwise $p_2, \ldots, p_{l+1}$ would be a shorter covering walk), and the same holds for p_{l+1}. Let $\sigma_{pq} = p_1, \ldots, p_l$ and $\sigma_{qp} = p_{l+1}, p_l, \ldots, p_2$. We will repeatedly request σ_{pq} and σ_{qp}. After the sequence $\sigma_{pq}\sigma_{qp}$, page p will be outside OPT's cache. After servicing the subsequence σ_{pq} (σ_{qp}), page q (page p) will be outside the cache. Therefore the first request of σ_{pq} (σ_{qp}) will always be a fault. $\square$

Corollary 2.16 *To determine the fault ratio of OPT on a given graph G is an NP-hard problem.*

Proof. The problem of deciding whether G has a hamiltonian path is NP-complete. Let $k = n - 1$ be the size of the cache. Since $f_{\mathrm{OPT}}(\Sigma(G)) = \frac{1}{k}$ if and only if G has a hamiltonian path, the problem whether G contains a hamiltonian path can be reduced to the problem of finding $f_{\mathrm{OPT}}(\Sigma(G))$. □

Proposition 2.17 *The fault ratio of OPT (with cache size k) on an access graph G can be computed in PSPACE. This holds even if the access graph G is directed.*

Proof. Theorem 2.13 says that the fault vector $\overline{f}_t$, that holds all information about the past requests needed to determine the future faults, can be stored in $O(n \log n)$ bits. Let us take the pair $(\overline{f}_t, s_t)$ (s_t is the t-th request) as a "configuration" of the request sequence at time t. Assume the configurations at times t and $t' > t$ are identical. Then, if

$$\sigma = s_1 s_2 \dots s_t s_{t+1} \dots s_{t'} \dots s_l$$

is a valid request sequence for G, then so is

$$\sigma^{(i)} = s_1 \dots s_t (s_{t+1} \dots s_{t'})^i s_{t'+1} \dots s_l$$

for any $i \geq 0$. Moreover, if OPT incurs m faults in the time interval $\langle t+1, t' \rangle$, then $\mathrm{OPT}(\sigma^{(i)}) = \mathrm{OPT}(\sigma) + (i-1)m$. We shall call the subsequence $\sigma' = s_{t+1} \dots s_{t'}$ a *cyclic subsequence* for the state $(\overline{f}_t, s_t)$.

Assume $\frac{m}{t'-t} > f_{\mathrm{OPT}}(G)$. Then for any b, there is an i such that $\mathrm{OPT}(\sigma^{(i)}) > |\sigma^{(i)}| f_{\mathrm{OPT}}(G) + b$. Therefore $\mathrm{OPT}(\sigma') \leq |\sigma'| f_{\mathrm{OPT}}(G)$.

By the definition of the fault ratio, for any $\epsilon > 0$ there is a request sequence σ such that $\mathrm{OPT}(\sigma) > (f_{\mathrm{OPT}}(G) - \epsilon)|\sigma|$. Let σ be the shortest sequence with this property. Since σ cannot be shortened, for any of its cyclic subsequences $\sigma' = s_{t+1} \dots s_{t'}$, $\mathrm{OPT}(\sigma') > (f_{\mathrm{OPT}}(G) - \epsilon)|\sigma'|$ holds. There must be a cyclic σ' that is not longer than the number of different configurations, which is bounded by $n(n-k+2)^n$. If we take ϵ such that $\epsilon n(n-k+2)^n < 1$ we obtain that there must be a cyclic subsequence σ' such that $\mathrm{OPT}(\sigma') = f_{\mathrm{OPT}}(G)|\sigma'|$.

Thus, to determine whether $f_{\mathrm{OPT}}(G) \geq f_?$ for some given $f_?$ it suffices to check if there is a cyclic subsequence σ' such that $\mathrm{OPT}(\sigma') \geq |\sigma'| f_?$. This can be easily done in nondeterministic polynomial space: we nondeterministically guess a configuration $(\overline{f}, p)$ and check whether

- $(\overline{f}, p)$ is reachable, i.e. there is a prefix sequence σ_0 which leads from the initial state $(\overline{f}_0, \varepsilon)$ to the state $(\overline{f}, p)$
- there is a cyclic sequence σ' leading from the state $(\overline{f}, p)$ to the same state that has cost to length ratio greater or equal to $f_?$.

These checks can be performed by nondeterministicaly guessing σ_0 and σ' one request at a time and by simulating OPT2 on them. By Savitch's theorem (see e.g.

[14]), PSPACE=NPSPACE. We may thus assume that we have a deterministic procedure that decides if $f_{\mathrm{OPT}}(G) \geq f_?$ in polynomial space.

Since $f_{\mathrm{OPT}}(G) = \frac{m}{l}$ for some m and l, where m is the cost incurred by OPT on some cyclic sequence σ' and l is the length of σ', we only have to try out all $m \leq l \leq n(n-k+2)^n$, check if $f_{\mathrm{OPT}}(G) \geq \frac{m}{l}$ and select the maximum ratio. This can clearly be done in polynomial space. $\square$

Note. A similar proposition holds for every model, in which the "state of the request sequence" can be stored in polynomial space, including the FM and FMR models introduced in Chapter 3. For these models, we would use the contents of the call stack together with the last request as the "state of the request sequence".

Example 2.18 Let G, $|G| = n$ be a graph and let the path $p_1p_2 \ldots p_n$ be a hamiltonian path in G. Let $k < n$. On a request sequence of the form $(p_1p_2 \ldots p_{n-1}p_np_{n-1} \ldots p_2)^*$, OPT has cost to length ratio $\frac{n-k}{n-1}$. According to Lemma 2.1, the fault ratio of OPT is not greater than this for any class Σ, hence $f_{\mathrm{OPT}}(\Sigma(G)) = \frac{n-k}{n-1}$ for any graph containing a hamiltonian path.

Conjecture 2.19 *Let G be an acces graph, $|G| = n$ and let k be the size of the cache. The fault ratio of OPT on G is $\frac{n-k}{n-1}$ if and only if G contains a hamiltonian path.*

Theorem 2.20 *Let G be a tree on $n = k+1$ vertices. The fault ratio of OPT on G is $\frac{1}{2(n-1)-l}$, where l is the length of the longest path in G.*

Proof. According to Theorem 2.15, it suffices to show that the shortest covering walk has length $2(n-1)-l$. Let σ be a covering walk on G with minimal length and let p and q be its start and end vertices. Every edge that doesn't lie on the unique path pq must occur in σ at least twice and every edge on the path pq must occur at least once. Hence, the length of the covering walk is at least $2(n-1-l)+l$.

To construct a covering walk of length $2(n-1)-l$, let p and q be two vertices such that the path pq has length l. Both p and q are leaves (otherwise we could construct a path longer than l). The covering walk is generated by a depth-first search from the vertex p. We select the order of visiting ancestor vertices such that q is the last vertex visited. The complete depth-first search traveses each edge twice. Since the last l edges in the traversal form a path from q to p, by omitting them we obtain a covering walk of length $2(n-1)-l$. $\square$

Theorems 2.15 and 2.20 suggest a question, whether it is possible for every graph G on $k+1$ vertices to find its spanning tree T, such that T has the same fault ratio as G. The negative answer to this question is witnessed by the graph on the figure 2.1.

The shortest walk covering all vertices has length 11 (which implies the fault ratio $\frac{1}{11}$), but its longest path has length 6, so none of its spanning trees has fault

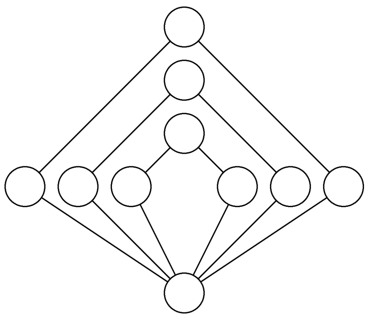

Figure 2.1: For each spanning tree T of the graph G, $f_{\mathrm{OPT}}(T) < f_{\mathrm{OPT}}(G)$.

ratio more than $\frac{1}{2(10-1)-6} = \frac{1}{12}$. To generalize this example, let G be a graph consisting of a cycles of length b sharing one common vertex. The shortest walk covering G has length $ab-1$, which gives us the fault ratio $\frac{1}{ab-1}$, but no path in G has length more than $2(b-1)$, therefore no spanning tree of G can achieve fault ratio higher than $\frac{1}{2a(b-1)-2(b-1)} = \frac{1}{2(a-1)(b-1)}$. If we take a and b large enough, we can construct graph with fault ratio almost two times greater than the fault ratio of any of its spanning trees.

Proposition 2.21 *For every $\varepsilon > 0$ there is a graph G, such that for every spanning tree T of G, the fault ratio of OPT on G is at least $(2-\varepsilon)$ times the fault ratio of OPT on T.*

Theorem 2.22 *Let G be a tree, $|G| = n$ and let l be the length of the longest path in G.*

a) If $l \geq 2(k-1)$, the fault ratio of OPT on G is $\frac{l+1-k}{l} \geq \frac{1}{2}$.

b) If $l \leq 2(k-1)$, the fault ratio of OPT on G is less or equal to $\frac{1}{2}$.

In order to prove the theorem, we first prove two lemmas.

Lemma 2.23 *Let T be a tree, let r be its vertex such that no path from r in T is longer than $k-1$. Assume that r is initially in the cache, remaining $k-1$ frames are empty. Then for every request sequence σ, $|\sigma| = l$ that starts with a request to a neighbor of r and ends with a request to r, OPT incurs at most $\frac{l}{2}$ faults on σ.*

Proof. We define a paging strategy that achieves the fault ratio $\frac{1}{2}$ (or better). It will always keep in the cache all pages on the (unique) path from r to the currently requested vertex p. We divide requests from the request sequence in two classes: those that increase the distance between r and the page pointer and those that decrease it. Requests in the second class never cause a fault, because the request is always to a page that lies on a path from the previous requested page to r. Since at the end of the sequence the distance from r is zero (r is the last request), both classes are of equal sizes and therefore OPT does not fault on more than half of the requests. □

Lemma 2.24 *Let T be a tree, let r be its vertex and let d be the distance from r to a most distant vertex, $d \geq k$. Assume that r is initially in the cache, remaining $k-1$ frames are empty. Then for every sequence σ, $|\sigma| = l$ that starts with a request to a neighbor of r and ends with a request to r, OPT incurs at most $(1 - \frac{k-1}{2d})l$ faults on σ. The shortest sequence that achieves this bound traverses the path to one of the most distant leaves and back and has length $2d$.*

Proof. Without loss of generality we can assume that r occurs in the request sequence only once, as the last request. Otherwise, we could split the request sequence into segments, each ending with a request to r and take just the segment that has the highest cost to length ratio. Now we proceed by induction on the length d of the longest path from r.

We follow by induction on d. As the base of the induction, let $d = k$. Let q be the first page requested in σ. If we delete the first and the last request of the request sequence σ, we obtain a sequence σ' operating on the subtree T' of T rooted at q. The first request of σ' is a neighbor of q, the last request is to the vertex q. In T', each path from q has length at most $k-1$, thus we know from Lemma 2.23 that the cost to length ratio of σ' is at most $\frac{1}{2}$. For σ to achieve a ratio higher than $\frac{1}{2}$, σ' must force OPT to evict the page r. The request to q and r are the only faults without a corresponding hit. The highest cost to length ratio of σ is thus achieved when σ' is as short as possible, but it must force OPT to evict r. Therefore it must cause at least $k-1$ faults and must be at least $2(k-1)$ requests long. A good candidate for σ' is a sequence that requests pages along the longest path. In summary, σ consists of $2k$ requests. In σ' there are $k-1$ faults, and two more are caused by the requests to q and r.

Let $d > k$. Again, we delete the first and the last requests from σ. By induction hypotesis, the shortened sequence σ' ($|\sigma'| = l - 2$) cannot cause more than $\frac{2d-k+1}{2d}(l-2)$ faults. To maximize the cost to length ratio of σ, σ' must force OPT to evict r and be as short as possible. This is again achieved by requesting all pages along the longest path. □

Proof of Theorem 2.22. Part b) follows from Lemma 2.23. Since no path in G is longer than $2(k-1)$, there must be a vertex r in G, such that every vertex is in distance at most $k-1$ from r.

For part a) first assume, that l is even. If we choose r to be the middle vertex of the longest path, no vertex is in distance more than $\frac{l}{2}$ from r and from Lemma 2.24 it follows that the fault ratio of OPT on G is at most $1 - \frac{k-1}{l}$.

For $l = 2m + 1$ odd, there are two vertices r_1 and r_2 joined by an edge such that each vertex is in distance at most m from the closer of the two vertices. By deleting the edge r_1r_2 we obtain two trees T_1 and T_2 rooted at r_1 and r_2. We break the request sequence into phases. A phase consists of requests to pages belonging to one tree. A phase is *short*, if no more than $k-1$ distinct pages have been requested. In this case we add the first request to page from the other tree (it must be r_1 or r_2) to the phase. In each short phase, OPT faults at most on half of the requests. A phase that is not short (a long phase) can begin with a request to a neighbor of r_1 (r_2), if it was preceeded by a short phase, or by a request to one of the roots r_1, r_2. In the former case we know from lemma 2.24 that the cost to length ratio of the phase is at most $\frac{2m-k+1}{2m}$.

If the long phase begins with a request to r_1 or r_2, we may assume that this request is a fault. If we must request at least k different pages without leaving the subtree, the request sequence that achieves the highest cost to length ratio again traverses the path from root to the most distant leaf and back. This can be proven in a fashion similar to the proof of lemma 2.24. This long phase is $2m+1$ requests and OPT incurs $2m+1-(k-1)$ faults during it, which gives in summary the fault ratio $\frac{2m+1-(k-1)}{2m+1} = \frac{l-k+1}{l}$. □

For the case when the length of the longest path is at most $2(k-1)$, Theorem 2.22 provides only an upper bound of $\frac{1}{2}$.

Conjecture 2.25 *Let G be a tree, $|G| = n$ and let l be the length of the longest path in G. If $l \leq 2(k-1)$, the fault ratio of OPT on G is $\frac{n-k}{2(n-1)-l}$.*

The shortest walk covering all vertices has length $2(n-1)-l$. We repeatedly request vertices along this walk. In each pass, $n-1$ different pages are requested, at most $k-1$ of them are in the cache, therefore OPT incurs at least $n-k$ faults in the pass.

Chapter 3

New models

As was already mentioned in section 2.4, only a little is known for the access graph model in the case of directed access graphs. Our intention was to work around this by introducing a new model, whose underlying structure is a directed graph (in fact, our models require the graph to be acyclic), for which it would be possible to obtain some non-trivial results.

The second and more important reason to introduce a new model is that the access graph model uses only a little information about the state of the program that generates requests to pages. In particular, the only information the paging system has is the "position of the instruction pointer" – the last requested page. In practice, the operating system should be able to collect more valuable information about the state of the algorithm without an additional large overhead. Namely, most modern programming languages prefer function calls to direct jumps into the code. The function call mechanism is generally implemented by the use of a stack: at each call, the address of the calling function is pushed onto the stack and at each return, the return address is popped from the stack. If the paging subsystem has access to the call stack, it can predict that the pages whose addresses are on the stack will be requested at some time in the future.

We propose two new models in this chapter: the FM and FMR model, defined in Section 3.2. Since we will be dealing with online algorithms, Section 3.1 gives an overview. Some properties and results for each of the models are given in sections 3.3 and 3.4. Notes on possible extensions of the models can be found in Section 3.5.

3.1 Online algorithms

Until now, we had no reason to take a closer look on online algorithms. Here we give a short overview of previous related work referred to in next sections. We discuss only deterministic algorithms.

The two commonly used and studied paging algorithms are *Least Recently*

Used (LRU) and *First In First Out* (FIFO). At a fault

LRU evicts the page, that was least recently requested and

FIFO evicts the page, that resides in the cache for the longest time.

It is often useful to view the optimal offline algorithm together with a generator of the request sequence as to an *adversary*, who plays against the online algorithm. The adversary tries to generate a costly request sequence for the algorithm, but it has to care that the sequence can be serviced with a small number of faults by an offline algorithm. If the online algorithm is deterministic (as in our case), the adversary knows the its state after each request of the sequence in advance. In particular it knows which pages are in the online algorithm's cache at each given time. The following two lemmas are classical results of Sleator and Tarjan [3]. The proof of the next lemma demonstrates the adversary style argument.

Lemma 3.1 *No deterministic online paging algorithm with cache size k has competitive ratio less than k (for the class of arbitrary request sequences).*

Proof. The adversary restricts itself to $n = k+1$ different pages. Since the online algorithm A is deterministic, the adversary at each time knows the contents of the A's fast memory. It always requests the page that is outside A's cache and forces A to fault at every request. This request sequence is then serviced by OPT. By Lemma 2.3, OPT faults at most once in every k requests. □

Algorithms *least recently used* (LRU) and *first in first out* (FIFO) are of both theoretical and practical interest.

Lemma 3.2 *Algorithms LRU and FIFO are k-competitive.*

Proof. The proof for both LRU and FIFO can be found in [3]. The k-competitiveness of LRU is also shown in Section 3.1.1. □

.

3.1.1 Marking algorithms

The class of marking algorithms has been defined by Karlin, Manasse, Sleator and Rudolph [11]. A marking algorithm proceeds in phases. At the beginning of a phase all pages are unmarked. Whenever a page is requested, it is marked. On a fault, the marking algorithm evicts an unmarked page (chosen by a rule specified by the algorithm) and brings in the requested page. A phase ends just before the first fault after all pages in the cache are marked. Observe that the definition of a phase is independent of the algorithm: the phase ends just before the request to the $(k+1)$-st distinct page requested in the phase. At this point all pages become unmarked and a new phase begins.

Lemma 3.3 *Every marking algorithm is k-competitive.*

Proof. Note that a marking algorithm faults on each page at most once in a phase. Thus, it incurrs at most k faults in the phase. By the following proposition, OPT faults at least once for each phase. $\square$

Proposition 3.4 *OPT incurs at least one fault for each phase.*

Proof. Divide the request sequence into segments starting with the second request of a phase and ending with the first request of the next phase. We claim that OPT faults at least once in each segment. The first request of a phase must be in the cache at the beginning of the corresponding segment. Since the segment together with the first request of the phase contain requests to $k+1$ different pages, at least one is outside the cache at the beginning of the segment and thus OPT faults. $\square$

Note that LRU is also a marking algorithm. Thus, by Lemma 3.3 it is k-competitive.

3.1.2 Online algorithms and tree access graphs

Our models are in a way extension of the undirected access graph model for the case when the access graph is a tree. The lower and upper bound on competitive ratio for this important class of graphs has been shown in [10].

For a tree T, let $l(T)$ denote the number of its leaves. Let G be a tree. We define $\lambda(G) = \max_{T \in \mathcal{T}_{k+1}} l(T)$, where $\mathcal{T}_{k+1}$ is the set of all connected subgraphs (trees) of G with $k+1$ vertices.

Lemma 3.5 *Let G be a tree, $|G| \geq k+1$. For each deterministic online paging algorithm A, $c_A(G) \geq \lambda(G) - 1$.*

Proof. Let $T \in \mathcal{T}_{k+1}$, $|T| = k+1$ be a connected subgraph of G that has $\lambda(G)$ leaves. The adversary restricts the requests to the $k+1$ vertices of T. It always requests vertices along the (unique) path to the page p that is outside A's cache. At most one leaf (namely p, if p is a leaf) lies on this path. Thus for every request to a leaf, A faults at least once.

The adversary keeps all interior vertices of T always in the cache. Since it faults only on leaves, we care only about requests to leaves and view it as having cache of size $l(T) - 1$ and servicing requests to $l(T)$ different pages. Thus, by Lemma 2.3, the adversary manages to fault at most once in every $\lambda(G) - 1$ requests to leaves of T. $\square$

Lemma 3.6 *If G is a tree, LRU is $(\lambda(G) - 1)$-competitive on G.*

Proof. The proof is rather complicated and is omitted. It can be found in [10]. □

The following definitions will be useful in the next sections. Let G be a directed acyclic graph. We say that vertex $r \in G$ is the *root* of G, if there is a directed path from r to any other vertex $p \in G$. Observe that each directed acyclic graph (DAG) has at most one root.

We extend the definition of λ to directed acyclic graphs. For a directed graph G, let $U(G)$ be its undirected version obtained by erasing directions of all edges. A directed acyclic graph G is said to be a (directed) tree, if G has a root and $U(G)$ is a tree. A vertex is a leaf of G iff it is a leaf of $U(G)$. Observe that there is at most one leaf with indegree 0 and outdegree 1; all other leaves have indegree 1 and outdegree 0. Let $\mathcal{T}_{k+1}$ be the set of all directed trees $T \subseteq G$ with $k+1$ vertices. We define $\lambda(G) = \max_{T \in \mathcal{T}_{k+1}} l(U(T))$.

3.2 New models

We propose two new models similar to the access graph model.

In the functional model, the request sequence simulates the pattern of calls of a program. Every vertex is identified with a function or procedure of the program, the root is identified with the main function (we assume that the code of each function fits exactly in one page). The computation is started by invoking the main function. It can then call any other function provided there is a directed edge from the root to the function called. Similarly, each function f can call a function g if there is an edge from f to g. At each call, a pointer to the the calling function is pushed on the stack. When the called function finishes, the pointer is retrieved from the stack and the control returns back to the calling function.

Let s_t be the page requested at time t. A request sequence is valid for a directed acyclic graph G with root r in the *functional model* (FM), if the first request $s_1 = r$ is to the root and all following requests are generated using step 1 (a forward step) or 2 (a return step):

1. Pick a vertex v, such that there is an edge $s_{t-1}v$. Push s_{t-1} on the call stack and let $s_t = v$. Or
2. Pop (i.e. delete) one entry from the top of the stack and let s_t be the popped page.

The call stack is initially empty.

The *call path* is the path from the root r to the currently requested vertex p that passes all vertices on the stack. If we pushed every vertex along the call path into an empty stack, its contents would be identical to the contents of the call stack.

The second model, the *functional model with return* (FMR), is a modification of the first one. The only difference is, that during a return step, it is possible to return more levels up.

2. Pop m ($m \geq 1$) entries from the top of the stack and let s_t be the last page popped.

In both models, the return step is possible only if the stack is not empty.

For a given directed acyclic graph G, we denote the class of valid request sequences in the functional model by FM(G) and the class of valid request sequences in the functional model with return by FMR(G).

3.3 Functional model

What is the "worst" graph for the FM model, i.e. for given n and k, which FM graph achieves the largest competitive ratio? Adding edges to a graph does not decrease its competitive ratio. If we add as many edges (preserving acyclicity) as possible, we always get a graph isomorphic to $G_n = (P_n, E_n)$, where $P_n = \{1, \ldots, n\}$ and E_n contains all edges $\overrightarrow{ij}$ for $i < j$.

Proposition 3.7 *For any deterministic paging algorithm A with cache size $k < n$, $c_A(G_n) \geq k - 1$.*

Proof. Take any sequence $\sigma = s_1 s_2 \ldots s_l$ of requests to pages $2, 3, \ldots, n$. We convert this sequence to a valid FM(G_n) sequence by interleaving it with requests to page 1: $1 s_1 1 s_2 1 \ldots 1 s_l$. On this sequence, no online algorithm can do better than to keep page 1 in the cache and use some online strategy on remaining $k - 1$ frames. The adversary also keeps page 1 in its cache and uses the optimal algorithm to service σ with remaining $k - 1$ frames. The $c_A(G_n) \geq k - 1$ lower bound follows now from Lemma 3.1. □

Example 3.8 LRU does not achieve the best possible competitive ratio on all graphs.

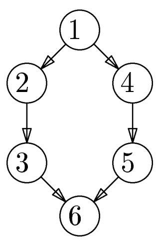

Figure 3.1: LRU is not optimal on all graphs in the FM.

For the graph on Figure 3.1, 1236(321456541236)* is a valid request sequence. Let the cache size $k = n - 1 = 5$. In each repetition of the segment in brackets, OPT faults twice (on requests to pages 3 and 5). LRU faults four times (at requests to pages 5,6,3 and 6). Therefore competitive ratio of LRU is at least 2. On the other hand, we design a marking algorithm (call it A) with competitive ratio 1.5.

A will never evict any of the pages 1,2 or 4. It distinguishes two kinds of phases. The first kind (a good phase) starts with a request (which is a fault) to page 6. In this case A evicts page 3 or 5 (whichever of them is not on the call stack). The adversary must request at least four pages, before it can request the evicted page and thus A incurs only one fault before the phase is over. The second kind of phases (a bad phase) begins with a request to page 3 or 5. W.l.o.g., let it be page 3. Our algorithm evicts page 5. If the adversary requests page 6 during the phase, A incurs only one fault during it. If the adversary requests page 5 instead, A incurs two faults, but in this case the next phase starts with a request to page 6 which is a good phase. Since after each phase in which A faults twice follows a phase with one fault, the total number of faults of A is at most 1.5 times the number of phases. By Proposition 3.4, OPT incurs at least one fault for each phase. □

We now describe the algorithm *FMleaf*. It is a marking algorithm. Let S be the set of all unmarked pages not lying on the call path. Recall that the call path is the path from the root to the currently requested vertex passing all vertices on the stack.

1. if $S \neq \emptyset$ evict a page $p \in S$ such that no path leads from p to any page $q \in S$.
2. if $S = \emptyset$, evict the first page on the call path that is in the cache.

The next lemma states a result similar to Lemma 3.6.

Lemma 3.9 *Let G, $|G| = n$ be a directed acyclic graph with root and let $k = n-1$ be the size of the cache. FMleaf is $\lambda(G)$ competitive on the class* $\mathrm{FM}(G)$ *of request sequences.*

Proof. We show that in a phase, FMleaf "builds" a directed tree T in which each vertex evicted during the phase is a leaf. Since the number of faults equals to the number of evicted vertices (no vertex is evicted twice in a phase), FMleaf incurs at most $l(T) \leq \lambda(G)$ faults in the phase. By Proposition 3.4, the adversary incurs at least one fault for each phase.

The tree T is built at the end of each phase. Let p_1 be the first page requested in the phase and let p_n be the first request of the next phase (thus, p_n is the one page not requested in the current phase). For each page $p \neq p_1$ that was requested in the phase, let e_p be the edge along which we first arrived to p. Let

$p \neq p_1$ be a page requested in the phase and let $s_t = p$ be the first request to p in the phase. Let $q = s_{t-1}$ the immediately preceeding request. We define e_p to be either $\overrightarrow{qp}$ or $\overrightarrow{pq}$, whichever edge is in G. Let T' be the graph consisting of all vertices p requested in the phase and all edges e_p for $p \neq p_1, p_n$. Obviously, $U(T')$ is connected. Since it contains $|T'| - 1 = k - 1$ edges, $U(T')$ is a tree.

OBSERVATION: For each pair of marked vertices p, q there is a marked vertex r_{pq} such that there is a path from r_{pq} to p and from r_{pq} to q and both paths pass only marked vertices and edges of T'.

Thus, T' is a directed tree. Let r' be the root of T'. We construct T by adding the vertex p_n and an edge to T'. We consider two cases:

1. $p_n \neq r$ is not the root of G. Then there is a page p, such that the edge pp_n is in G. Then, define $e_{p_n} = \overrightarrow{pp_n}$ and let $T = T' \cup \{p_n\} \cup \{e_{p_n}\}$.
2. $p_n = r$ is the root of G. Define $e_{p_n} = \overrightarrow{p_n r'}$ and let $T = T' \cup \{p_n\} \cup \{e_{p_n}\}$.

Let now p be a page that was evicted in the phase. We show that p is a leaf in T. Assume p is an inner node. Then there are two pages q_1 nad q_2 such that there are edges between p and q_1 and between p and q_2. It implies that there is a vertex q, such that $\overrightarrow{qp} = e_q$ or $\overrightarrow{pq} = e_q$.

Obviously, p has been marked before q was (or q was not marked in the phase). We claim that if p was evicted in the phase, then q was evicted before p. There is always only one page outside the cache, so if q is evicted, no other page is evicted until q is requested. Thus, q was marked before p was evicted which implies that p was not evicted during the phase.

To prove the claim assume that p is evicted at time t. If the edge $\overrightarrow{pq}$ is in G, by rule 1, FMleaf can evict p only if q is marked. If $\overrightarrow{qp} \in G$, q had to be immediately before p on the call path at the beginning of the phase and thus by rule 2, q should be evicted earlier than p. □

Theorem 3.10 *FMleaf is strongly competitive in the functional model, i.e. for each deterministic online paging algorithm* A, $c_{\text{FMleaf}}(\text{FM}(G)) = O(c_A(\text{FM}(G)))$ *for any directed acyclic graph with root* G.

Proof. We only sketch the main idea. Let g_i be the number of pages that are requested in the i-th phase but were not requested in the phase $(i-1)$ and T_i be the tree from proof of Lemma 3.9 built after phase i. In a fashion similar to the proof of Lemma 3.9 it can be shown that in the i-th phase, FMleaf faults only on leaves of T_i, except the leaf p_n which is the page requested as the first request of the $(i+1)$-st phase, and at most g_i other pages. Thus in the i-th phase, FMleaf incurs at most $g_i + \lambda(G) - 1$ faults. Summing over all phases

$$\text{FMleaf}(\sigma) \quad \leq \quad \sum_{i=1}^{a} g_i + a(\lambda(G) - 1)$$

$$OPT(\sigma) \geq \frac{1}{2}\sum_{i=1}^{a} g_i$$

where a denotes the number of phases. The second inequality is shown in the proof of Lemma 2.5. Taking into account that for each phase $g_i \geq 1$,

$$\text{FMleaf}(\sigma) \leq \lambda(G)\sum_{i=1}^{a} g_i \leq 2\lambda(G)\text{OPT}(\sigma)$$

and thus FMleaf is $2\lambda(G)$-competitive. By Lemma 3.5, no deterministic online algorithm is better than $\lambda(G) - 1$ competitive. □

Open Question 3.11 Prove or disprove: *FMleaf with cache size k is $\lambda(G)$-competitive on any directed acyclic graph with root G in the functional model.*

Open Question 3.12 Prove or disprove: *LRU is strongly competitive in the functional model, i.e. for each deterministic online paging algorithm A and any directed acyclic graph with root G, $c_{\text{LRU}}(\text{FM}(G)) = O(c_A(\text{FM}(G)))$.*

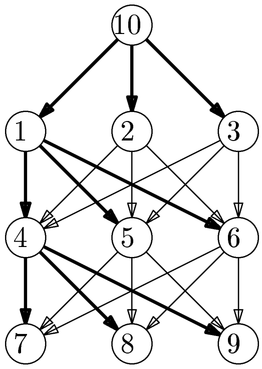

Figure 3.2: $c_{LRU}(G) > \lambda(G)$.

We note that LRU is not $\lambda(G)$ competitive on every graph G. Let G be the graph on Figure 3.2 and let the cache size be $k = |G| - 1$. Clearly, $\lambda(G) = 7$, but it can be shown that $c_{LRU}(G) \geq 9$.

3.4 Functional model with return

Let G be a directed graph with vertices $P = \{1, \ldots, n\}$, and edges $\overrightarrow{i(i+1)}$ for $1 \leq i < n$. Thus, G is a directed path.

Let G' be a graph obtained by adding backward edges $\overrightarrow{ij}$ for $n \geq i > j \geq 1$ to G. The (standard) access graph G' is equivalent to the FMRgraph G in the sense, that each request sequence from the class FMR(G) is a valid request sequence in $\Sigma(G')$ and each request sequence $\sigma \in \Sigma(G')$ can be made valid for FMR(G) by adding a short prefix (its length is bounded by $|G| - 1$). If the first request of σ

Figure 3.3: A FMR path and the equivalent directed access graph.

is to a page i, then the prefix would be $12\ldots(i-1)$. Thus the competitive ratio of the class FMR(G) equals to the competitive ratio of $\Sigma(G')$.

Observe that an equivalent access graph exists for every FMR graph which is a directed tree.

For the special case when G is a directed path, we have found a natural 2-competitive algorithm FMRPath. Let the request to page p be a fault:

1. if there is a page $q > p$ in the cache, evict such a page with the largest number
2. else evict the page with the smallest number.

Lemma 3.13 *Algorithm FMRPath is 2-competitive on a FMR graph that is a directed path.*

Proof. Let G be the directed path with vertices $1, 2, \ldots, n$ and let $\sigma \in \mathrm{FMR}(G)$ be a valid request sequence. We first modify σ by deleting some requests on which FMRPath hits. This way we do not decrease the number of faults of FMRPath and do not increase the number of faults of OPT. We will show te claim on the modified sequence.

Observe the use of the second rule. If a page p is requested with number graeater than any page in the cache, we can easily show that FMRPath has in its cache pages $p-k, p-k+1, \ldots p-1$. We take the longest subsequence of σ ending just before the request to p consisting of requests to pages $p-k, p-k+1, \ldots p-1$ only and delete from it all but the first occurence of each page.

We divide this modified sequence into phases. Each phase begins with a sequence of hits and faults serviced using the first rule. At the end of a phase there is a sequence of (at least one) faults serviced using the second rule. The phase ends just after the last such fault. We refer to all requests but the faults serviced using the second rule as the first subphase, the rest is the second subphase. We define a potential function Φ to be the maximum of the numbers of pages in the cache of FMRPath. Let $\mathrm{OPT}(\sigma_i)$ and $\mathrm{FMRPath}(\sigma_i)$ be the cost incurred in the i-th phase by OPT and FMRPath respectively. We claim that in a phase

$$\mathrm{FMRPath}(\sigma_i) \leq 2\mathrm{OPT}(\sigma_i) - \Delta\Phi_i \tag{3.1}$$

where $\Delta\Phi_i$ is the difference in the potential at the end of the i-th and the $(i-1)$-st phases.

In the first subphase, each fault incurred by FMRPath is serviced by evicting the vertex with the largest number. Therefore, Φ decreases by at least 1. In the second subphase, each request increases Φ by one and causes a fault of FMRPath. Let a be the number of faults incurred by FMRPath in the first subphase of the i-th phase, b be the length of the second subphase and p be the last page requested in the phase. The last $a + k$ requests were to $b + k$ different pages (namely $p-b-k+1, p-b-k+2, \ldots, p$). This means that OPT incurred at least b faults.

$$\begin{aligned} 2b &\leq 2\mathrm{OPT}(\sigma_i) \\ \mathrm{FMRPath}(\sigma_i) &\leq a + b \\ \Delta\Phi_i &\leq b - a \end{aligned}$$

By adding these inequalities and rearranging we obtain the claim 3.1. If we sum the inequalities 3.1 over all phases and realize that Φ is bounded ($k \leq \Phi \leq n$) we obtain the statement of the lemma. Further, if we assume that both algorithms start with cache contents $1, 2, \ldots, k$, we have $FMRPath(\sigma) \leq 2OPT(\sigma)$ for every $\sigma \in \mathrm{FMR}(G)$, since the start potential is less than or equal to the final potential. $\square$

The bound in the above lemma is not tight. For $n = k + 1$ we have a slightly stronger result:

Proposition 3.14 *FMRPath has competitive ratio $2\frac{n-1}{n}$ if the cache size is $k = n - 1$.*

Proof. It is easy to see that FMRPath at a fault evicts only page 1 or n. Let FMRPath faults on some page p and evicts page q. In the segment of the request sequence σ between requests to p and q FMRPath does not fault. Hence, deleting any requests from this segment does not decrease the number of faults of FMRPath; it cannot increase the number of faults of OPT. Therefore we will assume that it is not possible to delete requests from any segment between two faults of FMRPath. This assumption allows only one possible request sequence, since after request to page n, page 1 is evicted, so the next request is to page 1. Similarly, after request to page 1, page n is evicted and the next requests (to pages $2, 3, \ldots, n$) are also uniquely determined. On this sequence, FMRPath faults twice every n requests, OPT faults once every $k - 1$ requests. $\square$

The FMRPath algorithm can be extended to service request sequences $\sigma \in \mathrm{FMR}(G)$ where G is a directed tree. Let T denote the subgraph of G induced by all vertices residing in the cache at time $t - 1$ and the t-th page requested $p = s_t$. The algorithm FMRTree works as follows. On a fault it evicts a page according to these two rules:

1. if $U(T)$ is connected, evict the least recently used leaf of $U(T)$

2. else evict the least recently used vertex of T_t with outdegree 0 that does not lie in the same component of $U(T)$ as the current request s_t.

The competitive ratio of FMRTree has yet to be investigated.

3.5 Possible extensions

In the end we would like to point out a few questions that could be addressed in the future research.

- What happens, if we allow the underlying graph in the FM (FMR) model to contain cycles? With this modification it is reasonable to impose a limit on the size of the call stack to prevent unbounded number of recursive calls without corresponding returns. An alternative is to require that the call stack be empty at the end of the request sequence.
- A valid request sequence for a graph G in the FMR model can be alternatively defined as follows. Let the call stack be initially empty and let the first request s_1 of the request sequence be to the root r. Consequent requests are genereted using the following rules:

  1a. Pick a vertex v, such that there is an edge $s_{t-1}v$. Push s_{t-1} on the stack and let $s_t = v$.

  1b. Same as 1a, but do not push s_{t-1} on the stack.

  2. Pop one entry from the top of the stack and let s_t be the popped page.

  Thus for each page the adversary must decide in the forward step, whether the previous page shall be requested in a corresponding return step or whether it shall be skipped. According to the original definition, this decision was deferred to the corresponding return step. In our current setting this change in definition makes no difference, since the online algorithm cannot access the call stack.

  Assuming the online algorithms can "see" the call stack, is there a difference in their performance using the original and the modified definition?
- In practice, the code of many functions does not fit into one page. Conversely, one page can hold the code of more than one function. These concerns are not addressed in the FM and FMR models.

Bibliography

[1] Amos Fiat and Gerhard J. Woeginger (Eds.). *Online Algorithms: State of the Art.* Springer-Verlag, 1998

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

[3] Daniel D. Sleator and Robert E. Tarjan. *Amortized Efficiency of List Update and Paging Rules.* Communications of the ACM, 28:202–208, February 1985.

[4] Marek Chrobak and John Noga. *LRU is Better than FIFO.* SODA '98, pp. 78–81.

[5] Amos Fiat and Manor Mendel. *Truly Online Paging with Locality of Reference (Extended Abstract).* FOCS '97, pp. 326–335.

[6] Amos Fiat and Anna Karlin. *Randomized and Multipointer Paging with Locality of Reference.* STOC '95, pp. 626–634.

[7] Neal E. Young. *On-Line File Caching.* Technical Report PCS-TR97-320, Darmouth College, Department of Computer Science, 1997.

[8] Amos Fiat, Richard M. Karp, Michael Luby, Lyle McGeoch, Daniel D. Sleator and Neal E. Young. *Competitive paging algorithms.* Journal of Algorithms, 12:685–699, 1991.

[9] Sandy Irani, Anna R. Karlin and Steven Phillips. *Strongly Competitive Algorithms for Paging with Locality of Reference.* SODA '92, pp. 228-236.

[10] Allan Borodin, Sandy Irani, Prabhakar Raghavan and Baruch Schieber. *Competitive paging with locality of reference.* STOC '91, pp. 249-259.

[11] Anna R. Karlin, Mark S. Manasse, Larry Rudolph and Daniel D. Sleator. *Competitive Snoopy Caching.* Algorithmica, 3:79–119, 1988.

[12] Thomas H. Cormen, Charles E. Leicerson and Ronald L. Rivest. *Introduction to Algorithms.* McGraw-Hill, 1990

[13] Reinhard Diestel. *Graph Theory, Second Edition.* Graduate Texts in Mathematics, Volume 173. Springer-Verlag, 2000

[14] Jozef Gruska. *Foundations of Computing.* Thomson Computer Press, 1997.