Journal of Graph Algorithms and Applications

http://www.cs.brown.edu/publications/jgaa/

vol. 2, no. 6, pp. 1–22 (1998)

Memory Paging for Connectivity and Path Problems in Graphs

Esteban Feuerstein

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
and
Instituto de Ciencias
Universidad de General Sarmiento
Argentina.
efeuerst@dc.uba.ar

Alberto Marchetti-Spaccamela

Dipartimento di Informatica e Sistemistica
Università di Roma "La Sapienza"
Italy
alberto@dis.uniroma1.it

Abstract

We extend the Paging Problem to the case in which the items that are stored in the cache memory represent information about a graph. We propose on-line algorithms for two different connectivity problems in this context, for particular classes of graphs and under different cost assumptions. In the Path-paging problem we assume that the cache contains edges of the graph and queries to be answered are of the kind "report a path from i to j"; to answer the query it is necessary to have in memory all the edges of a path from i to j. In this case the answer to a query is not a single piece of information stored in memory. In the Connectivity problem the edges of the transitive closure of a given graph are stored in memory and we want to answer connectivity queries. In order to positively answer connectivity queries of the type "is i connected with j?", it is possible to answer the query even if the cache does not contain the edge (i, j). Most of our algorithms are optimal and fairly simple.

Communicated by R. Tamassia: submitted October 1994; revised September 1998.

This work was partly supported by EC project DYNDATA under program KIT, by EC Esprit Long Term Research project ALCOM IT under contract 20244, by UBA projects EX070/J "Algoritmos eficientes para problemas on-line con aplicaciones" and EX209 "Modélos y Técnicas de Optimización Combinatoria" and by Italian Ministry of Scientific Research Project 40% "Algoritmi, Modelli di Calcolo e Strutture Informative". A preliminary version of this work has been presented at the Fourth International Symposium on Algorithms and Computation ISAAC'93, Hong Kong, December 1993.

1 Introduction

The input of an on-line problems is, in general, a sequence of requests each of which is to be served without knowing future requests. Such problems arise, for example, in memory management where an algorithm has to decide on-line which page of memory must be evicted from fast memory when a page fault occurs. Namely, in the Paging Problem [18], a fast memory (the *cache*) can contain at most a constant number of pages and a sequence of page requests is presented; if the requested page is in the cache, then the request can be answered with zero cost; otherwise, a page fault occurs and it is necessary to move the page from secondary memory to the cache paying a unit cost. The decision on which page to evict must be taken on-line, without knowing future requests.

In this paper we study two applications of the paging problem to graph problems. The considered problems extend the paging problem to the case in which each piece of information, eventually combined with other pieces, can be used to infer information not directly present in the cache. As an example, in the Connectivity problem the edges of the transitive closure of a given graph are stored in memory and we want to answer connectivity queries. In order to positively answer connectivity queries of the type "is i connected with j?", it is possible to answer the query even if the cache does not contain the edge (i,j); in fact it is sufficient that the cache contains a positive information about the connectivity of i and vertex h and of h and j, where h is any vertex.

The other problem that we consider is the Path paging problem: we assume that the cache contains edges of a connected graph and queries to be answered are of the kind "report a path from i to j"; to answer the query it is necessary to have in memory all the edges of a path from i to j. In this case the answer to a query is not a single piece of information stored in memory. Note that there might be many possible paths and, therefore, many possible answers.

In their seminal paper [18], Sleator and Tarjan introduced *amortized analysis* to analyze the performance of algorithms for on-line problems; they compare the cost of an on-line algorithm for answering a sequence of requests with the cost of the optimal algorithm that knows the whole sequence of requests. An algorithm is said to be *c-competitive* [12] if the cost afforded by it to serve any sequence of requests is at most c times the cost charged to the optimal algorithm plus a constant. Neither the constant c nor the additive constant depend on the particular input sequence.

During the last years considerable attention has been devoted to competitive analysis of on-line algorithms, particularly to extend and generalize the Paging Problem. This includes the weighted version of the paging problem [17, 20], the problem of maintaining caches in a multiprocessor system [12], the k-server problem [7, 8, 10, 11, 13, 15]. All above extensions to the paging problem consider increasingly more complex models, but share essentially the same fundamental "individuality" property: in the case of paging, each request to a page of memory requires *that* page to be present in the cache; in the k-server problem one server must be present in *one* specified location to serve the request.

The problems that we consider are a special case of the more general notion of Metrical Task Systems [6] that allows to model a great variety of on-line problems. However, the generality of that approach implies that the results that have been proved are rather negative results. In fact, competitive algorithms with small constant competitiveness coefficients have been found for different versions of the paging and the k- server problem [7, 8, 10, 11, 12, 13, 15, 17, 18, 20]. On the other side, a lower-bound on the competitiveness of any on-line algorithm for Metrical Task Systems has been proved that is linear in the number of different states [6]. For the problems considered in this paper, the number of states is exponential in the size of the cache. However, the competitiveness coefficients that we obtain are polynomial in the size of the cache.

Besides their theoretical interest, the problems that we consider are motivated by the memory management problem of data structures for very large graphs (such that it is not possible to store in main memory neither the graph nor its transitive closure). We are interested in analyzing algorithms that use a small fast memory with the goal of reducing the number of accesses to the slow memory. We believe that the proposed algorithms might be useful in understanding and optimizing the cost of accessing secondary memory when dealing with very large graphs. Another possible motivation concerning communication in a network of processors will be discussed in section 4.

We are not aware of previous work on this subject; however a considerable amount of work on related matters can be found in the literature. Borodin et al. [5] considered the paging problem in the case in which the sequence of requests follows a pattern given by a previously known access graph. Nodine et al. [14] studied the speed-up that may be obtained using redundancy and efficient blocking in the case of searching in particular classes of graphs that are too large to fit in internal memory. The difference is that in our problem all queries *refer* to an underlying graph, but the sequence of requests does not follow any specified pattern. Attention has been also devoted [16, 19] to efficiently storing particular data structures in secondary memory in a dynamic environment, that is, when the underlying data structure is updated dynamically.

Other related problems have been considered by Aggarwal et al. [1, 2, 3]. These works introduced different models of hierarchical memories and analyzed the behavior of algorithms for different problems in those models. Our problem is different from the cited works in the sense that we do not have to minimize the number of accesses to secondary memory necessary to solve a certain off-line problem but to answer in an on-line fashion an arbitrarily long sequence of queries.

In section 2 the Path paging problem is defined and studied. We show that any on-line algorithm to be competitive for this problem against and adversary with cache of size k needs a cache of size at least $\lfloor \frac{3}{4}k^2 - \frac{k}{2} + 1 \rfloor$. We also study this problem in the particular case of trees, giving lower and upper bounds for the competitiveness of deterministic and randomized algorithms. Depending on the cost model considered, lower and upper bounds match or differ only by a constant factor.

In section 3 we consider the Connectivity problem. We show that any on-line algorithm must have a cache that is at least $\Omega(k^2)$ in order to be competitive against an adversary with a cache of size k; we also prove a lower bound of k for the competitiveness of any on line algorithm with any cache size against the same adversary. We also show a strategy for this problem that is a constant factor away from optimal.

Section 4 presents optimal solutions to the connectivity problem in the particular case of connected graphs, that allows to model the service of requests in high-speed computer networks. Finally, conclusions and open problems are presented in section 5.

1.1 Competitive analysis

An on-line algorithm for a given problem is c-competitive [12] if the cost afforded by it to serve *any* sequence of requests is less than c times the cost charged to the optimal algorithm for that sequence plus a constant that does not depend on the chosen sequence. Let $C_A(\sigma)$ denote the cost afforded by algorithm A to serve an input sequence σ, and let OPT be the optimal off-line algorithm. Formally, we have the following definition:

Definition 1.1 *An on-line algorithm A is* c-competitive *if there exists a constant d such that for any input sequence σ,*

$$C_A(\sigma) - c * C_{OPT}(\sigma) \leq d.$$

If $d = 0$ then A is strongly c-competitive*. The* competitive ratio *of algorithm A is the infimum over c such that A is c-competitive.*

As it is usually done in competitive analysis, we will compare on-line strategies with an *adversary* that must serve the same sequence of requests with his own cache, but who knows (in fact, who chooses) the entire request sequence in advance. Proving that the cost afforded by an on-line algorithm is no more than c times the adversary's cost on the same sequence of requests implies that the on-line strategy is c-competitive.

For the paging problem it has been shown [18] that a simple First In First Out (FIFO) rule is optimal, i.e. it obtains the best possible competitive ratio. In fact, assuming that the algorithm and the adversary have the same memory of size k, then FIFO achieves a competitive ratio of k; and k is also a lower bound.

In the case of randomized strategies, different kinds of adversaries have been proposed. The most used adversary is the *oblivious* adversary that generates the input sequence of requests and then submits it to the on-line algorithm. Other possible adversaries include the *adaptive-on-line* and the *adaptive-off-line* adversaries [17]. In the following we will restrict our attention to the oblivious adversary.

Note that deterministic and randomized lower-bounds for the competitive ratio of on-line algorithms for the Paging problem can be immediately extended to the problems considered in this paper; namely k is a lower bound for deterministic algorithms and H_k (the k-th harmonic number) is a lower bound for randomized algorithms against an oblivious adversary [9].

For many on-line problems, in particular paging problems, the performance of an on-line algorithm with respect to an adversary having less resources has been considered. This type of analysis can provide an insight on the behavior of the algorithm when the resources assigned to it may change. For example, in [18, 17, 20] different strategies for the Paging problem have been proved to be $\frac{K}{K-k+1}$-competitive when on-line algorithms are assigned a cache of size $K \geq k$; it has also been proved that this value is optimal.

In the following we denote by K the size of the on-line algorithm's cache and by k that of the adversary's; when they are equal we denote the size of both by k.

Some of our proofs use the standard technique of the *potential function* [18], that is a function that maps every pair D of configurations of the cache of the on- line algorithm and of the adversary's to a value $\Phi(D)$. In this model, the amortized cost of an operation is given by $t+\Phi(D')-\Phi(D)$, where t is the actual cost of the operation and D and D' represent the configurations before and after the execution of the operation respectively. We will use the following lemma, whose standard proof is left to the reader. This or similar lemmas have been used in previous works on on-line algorithms (see for example [8]).

Lemma 1.1 *Let C_{ADV} and C_{ALG} denote the total costs charged respectively to an adversary and on-line algorithm ALG to serve a sequence of requests. Suppose that Φ, $\Phi \geq 0$, is a potential function with value Φ_0 in the initial configuration such that*

1. *when the adversary makes a move, the increment in the potential is not more than α times the cost paid by the adversary, and*
2. *when the on-line algorithm serves a request, Φ decreases by at least β times the cost paid by the algorithm to serve the request*

Then $C_{ALG} \leq (\alpha/\beta)C_{ADV} + \Phi_0$, and hence ALG is (α/β)-competitive.

1.2 Results of the paper

When an algorithm cannot answer a query with the information present in its fast memory, it must search for the information in secondary storage. It is reasonable to assume that it will search for a shortest path between the requested vertices. However, the on-line algorithm does not know which path will be more helpful for answering future queries. We consider three different cost models:

1. *Full-cost model*: when a path joining a and b is not present in cache, charge the algorithm the length (i.e. the number of edges) of a shortest path from a to b.
2. *Partial-cost model*: when a path joining a and b is not present in cache, charge the algorithm the number of edges it brings into the cache to have a path between the query vertices.
3. *0/1-cost model*: when a path joining a and b is not present in cache, charge the algorithm a unit cost.

Tables 1 and 2 present a summary of the results of the paper (H_k denotes the k-th harmonic number).

2 Path-paging

The *Path-paging problem* is defined as follows: given an undirected connected graph, a query *Path*(a, b), means "give a path joining vertices a and b"; we consider the case when a sequence of such queries must be answered in an on-line fashion: the i-th query must be answered before the following queries are presented. The goal is to minimize the number of accesses to secondary memory performed in order to serve a sequence of requests.

We analyze the behavior of algorithms that use two levels of memory: a small and fast level (the *cache*) consisting of a constant number of edges and a secondary level of memory that allows to store the whole graph. The only stored information in the cache is a set of edges; no further information is stored.

Table 1: Summary of results - Path-paging problem

Problem	Full cost	Partial cost	0/1 cost
Lower bound for arbitrary graphs, $K < \frac{3}{4}k^2$	unbounded	unbounded	unbounded
Upper bound for arbitrary graphs, $K = \frac{3}{4}k^2$	$\lceil \frac{3}{4}k^2 \rceil$	$\lceil \frac{3}{4}k^2 \rceil$	k
Lower bound for trees	$\lfloor k^2/4 \rfloor + 1$	k	k
Upper bound for trees	$(k^2+k)/2$	k	k
Lower bound for trees, randomized	$\frac{k}{2}\ln k$	H_k	$\lfloor \frac{k}{e} \rfloor + 1$
Upper bound for trees, randomized	$2kH_k$	$2H_k$	k

Table 2: Summary of results - Connectivity-paging and Link-paging problems

Problem	
Lower bound for arbitrary graphs, $K < \frac{k^2}{2} - 3\left(\frac{k}{2}\right)^{\frac{5}{3}} + O(k^{\frac{4}{3}})$	unbounded
Lower bound for arbitrary graphs, any K	k
Upper bound for arbitrary graphs $K \approx \frac{k^2}{2} + k$	$\frac{k^2}{2} + k$
Lower bound for complete graphs (Link-Paging)	$\frac{K}{K-k+1}$
Upper bound for complete graphs (Link-Paging)	$\frac{K}{K-k+1}$

An algorithm serves a query *Path*(a, b) without cost if its cache already contains all edges of some path connecting a and b. If such edges are not present, then a set of edges forming a shortest-path from a to b is provided to the on-line algorithm, and a cost is charged to it [1]. Clearly the problem makes sense only if the diameter of the graph (i.e. the maximum distance among all pairs of vertices) is bounded by the size of the cache.

2.1 Path-paging for arbitrary graphs

We first show that no on-line deterministic algorithm can be competitive unless the cache of the algorithm is sufficiently larger than the adversary's cache; the bound holds in the case of arbitrary connected graphs and for all cost models.

Theorem 2.1 *No deterministic strategy for the Path-paging problem for arbitrary graphs with a cache of size $K < \lfloor \frac{3}{4}k^2 - \frac{k}{2} + 1 \rfloor$ can achieve a bounded competitive ratio against an adversary with cache size k, under all cost models.*

Proof: Consider a sequence of $(K+1)k$ requests to one-edge paths

$$e_{11}, \ldots, e_{1k}, e_{21}, \ldots, e_{2k}, \ldots, e_{(K+1)1}, \ldots, e_{(K+1)k}$$

such that each subsequence $e_{i1}, \ldots, e_{ik}$ forms a path from vertex v_{i1} to vertex $v_{i(k+1)}$ (the graph corresponding to one such subsequence is depicted in figure 1 (a)).

From now on we suppose k is odd, if k is even the proof is similar. Since the size of the on-line algorithm's cache is K, after such a sequence there exists at least one i such that the on-line strategy has none of $e_{i1} \ldots e_{ik}$ in the cache. This holds whichever were the initial configurations of the on-line algorithm and that of the adversary. As the adversary knows the exact value of such i, he can serve the sequence of requests as follows (for simplicity of notation, we will denote all edges e_{ij} and vertices

[1] We assume that on-line algorithms can control what kind of path to search for, and for this reason we assume that they look for shortest paths. In fact such paths provide the required information with a "minimum" use of the cache. Moreover, without this assumption, competitiveness results would be impossible. Note however that on-line algorithms have no way of knowing whether the retrieved information will be useful for future requests

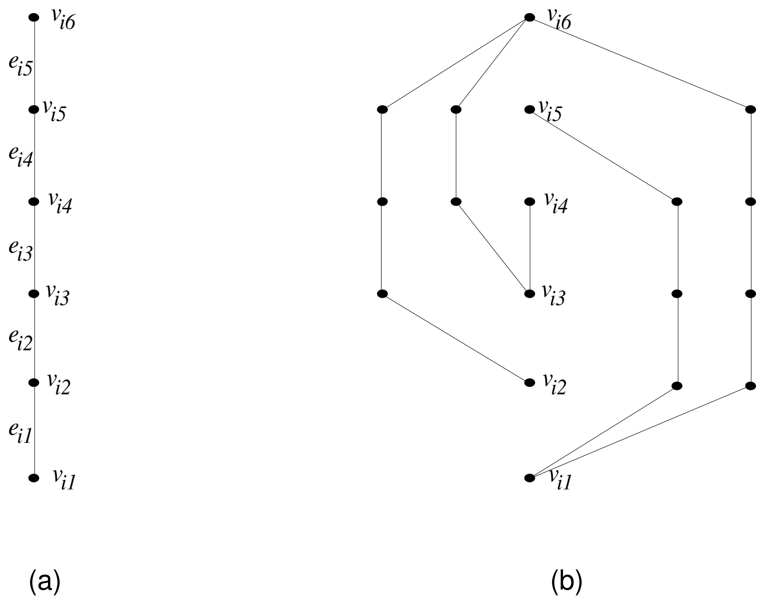

Figure 1: Example for Theorem 2.1, with $k = 5$

v_{ij} as $\hat{e}_j$ and $\hat{v}_j$ respectively). The subsequence $e_{11} \dots e_{1k} \dots e_{(i-1)1} \dots e_{(i-1)k}$ can be served in any way; for $\hat{e}_1 \dots \hat{e}_k$ the adversary puts each edge in a different position of the cache, so as to keep all edges of the subsequence. Let $\kappa = \frac{k+1}{2}$. For the remaining queries of the sequence, the adversary pays for every query, caching all the edges to the same position of his cache, namely the position that contained edge $\hat{e}_\kappa = (\hat{v}_\kappa, \hat{v}_{\kappa+1})$.

After the last query of the sequence the adversary may ask again for $Path(\hat{v}_\kappa, \hat{v}_{\kappa+1})$, and hence be ready to answer without paying the following queries:

$$Path(\hat{v}_1, \hat{v}_{k+1}), Path(\hat{v}_1, \hat{v}_k), \dots, Path(\hat{v}_1, \hat{v}_{\kappa+2})$$

and then

$$Path(\hat{v}_2, \hat{v}_{k+1}), Path(\hat{v}_3, \hat{v}_{k+1}), \dots, Path(\hat{v}_\kappa, \hat{v}_{k+1}).$$

If, for each one of these queries, the on-line algorithm is provided a shortest path of the same length but disjoint from the path stored by the adversary and from previously requested paths (see figure 1 (b)), then we have that the total length of the requested paths is

$$1 + \sum_{j=\kappa+1}^{k} j + \sum_{j=\kappa}^{k-1} j = \frac{3}{2}k + \frac{3}{2} + 2\sum_{j=\kappa+1}^{k-1} j = \frac{3}{4}k^2 - \frac{k}{2} + \frac{3}{4}$$

Since the total size of the on-line algorithm's cache is $K < \lfloor \frac{3}{4}k^2 - \frac{k}{2} + 1 \rfloor$, then there will be always one query that can be answered by the adversary with no cost for which the on-line algorithm has to pay. The theorem follows. □

We now analyze the behavior of the FIFO algorithm that, each time a new edge has to be brought into the cache, evicts the edge that has been present in the cache for the longest time.

Theorem 2.2 *For the Path-paging Problem on arbitrary graphs, with a cache of size* $\lceil \frac{3}{4}k^2 \rceil$*, and against an adversary with cache of size* k*, FIFO is:*

- $\lceil \frac{3}{4}k^2 \rceil$*-competitive in the full-cost model*
- $\lceil \frac{3}{4}k^2 \rceil$*-competitive in the partial-cost model*

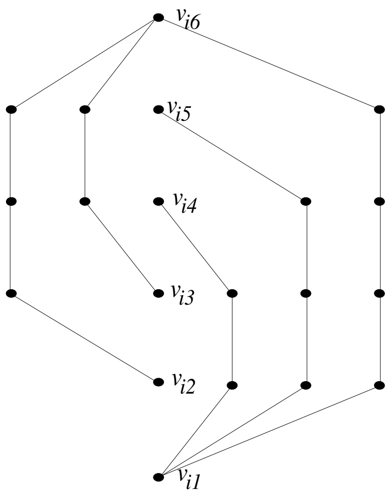

Figure 2: Example for Theorem 2.2, with $k = 5$

- *k-competitive in the* $0/1$-*cost model.*

Proof: Without loss of generality we assume that all queries in the input produce a fault to FIFO. We divide the sequence of queries in phases as follows: the first phase starts with the first query that implies a fault of FIFO, and a phase starting with query σ_i ends with query σ_j such that j is the minimum integer such that the paths referred in the queries $\{\sigma_i, \sigma_{i+1}, \ldots, \sigma_{j+1}\}$ require more than k edges to be answered. In other words, any subgraph of the underlying graph in which all the pairs referred in the queries of a phase are connected has strictly more than k edges. It is obvious that during a phase the adversary faults at least once, since it is not possible to answer the first query of each phase with the same set of edges that allows to answer all the queries of the previous phase. Hence the adversary's amortized cost is at least 1 for each phase, in all cost models.

Let S be the sum of the lengths of all the paths that produce a fault to FIFO during a phase. By showing that S is bounded by $\lceil \frac{3}{4}k^2 \rceil$, we prove the claim for the full and partial cost models.

Note that, during a phase, at most k different requests can produce a fault for FIFO (each fault will provide the information corresponding to at least one edge of the adversary). This immediately implies the claim for the $0/1$-cost model, as the cost of each request in that cost model is at most 1.

We show that S is maximized for a particular sequence of requests that the adversary can construct whenever the edges of its cache form a simple path (an example of which is depicted in Figure 2, for $k = 5$). Let us suppose k is odd (the case in which k is even is similar and it is omitted), and, as let $\kappa = \frac{k+1}{2}$. The adversary holds a simple path from v_1 to v_{k+1}, and the queries are of the form $Path(v_1, v_j), \kappa+1 \leq j \leq k+1$ and $Path(v_j, v_{k+1}), 2 \leq j \leq \kappa$. In this case the total length of the requested paths (and therefore the maximum cost incurred by FIFO during a phase) is

$$S = \sum_{j=\kappa}^{k} j + \sum_{j=\kappa}^{k-1} j = k + 2\sum_{j=\kappa}^{k-1} j = \frac{3}{4}k^2 + \frac{1}{4} = \lceil \frac{3}{4}k^2 \rceil.$$

We now show that the above example maximizes S over all possible sets of paths to be requested during a phase.

Since all the connectivity information given by a cyclic graph is present in a spanning forest of the graph, without loss of generality we may suppose that the adversary has no cycles in its cache. Moreover, we assume that all the paths in the spanning forest are shortest paths in the underlying graph, and that the paths provided to FIFO are always disjoint from those of the adversary. This last assumption can only increase FIFO's cost.

We will first prove that, given that the adversary has a spanning forest, all queries refer to a vertex and its furthest leaf, and that no internal vertex appears in two different queries. Then we show that if each tree of the spanning forest is a simple path then S is maximized, and finally that the best choice for the adversary is given by having only one connected component (that is, one simple path). Let $d(x,y)$ be the distance between x and y.

1. All queries refer to a vertex and its furthest leaf. Suppose there is a query involving two internal vertices n_i and n_j. It is obvious that it could be substituted by a query involving one of the internal vertices (say n_i) and a leaf ℓ with $d(n_i,\ell) > d(n_i,n_j)$. Moreover, any query of the form $Path(n,\ell)$ may be replaced with a query $Path(n,\ell')$, where n is an internal vertex and ℓ' is the furthest leaf from n.

2. No internal vertex appears in two different queries. Suppose that there is an internal vertex n and two different leaves ℓ_1 and ℓ_2 such that the queries $Path(n,\ell_1)$ and $Path(n,\ell_2)$ appear in the input sequence, in that order. We consider two cases:

   - $d(\ell_1,\ell_2) \geq d(n,\ell_2)$. Clearly, by asking $Path(\ell_1,\ell_2)$ instead of $Path(n,\ell_2)$, we would obtain a sequence with at least the same S. This substitution is possible unless FIFO has already a path from ℓ_1 to ℓ_2 when $Path(n,\ell_2)$ is requested. But this would imply that the query $Path(n,\ell_2)$ would not produce a fault (because FIFO has also a path from n to ℓ_1).
   - $d(\ell_1,\ell_2) < d(n,\ell_2)$. In this case n does not belong to the path from ℓ_1 to ℓ_2. Since n is an internal vertex, there exists a leaf ℓ_3 such that $d(\ell_1,\ell_3) > d(\ell_1,n)$, $d(\ell_2,\ell_3) > d(\ell_2,n)$. Moreover, the sequence does not contain both requests $Path(\ell_1,\ell_3)$ and $Path(\ell_2,\ell_3)$ (otherwise, the last of the four paths to be requested would not produce a fault, a contradiction).

     Two different cases must be considered:

     – $Path(\ell_1,\ell_3)$ is not requested. If $Path(n,\ell_1)$ can be replaced by $Path(\ell_1,\ell_3)$ we are done. Otherwise, when $Path(n,\ell_1)$ is requested FIFO has already a path from ℓ_1 to ℓ_3 that does not pass through n. But then, after the request $Path(n,\ell_1)$ FIFO has a path from n to ℓ_3. If before the request $Path(n,\ell_2)$ FIFO had also a path from ℓ_2 to ℓ_3, the request $Path(n,\ell_2)$ would not produce a fault, a contradiction. Hence, FIFO had not a path from ℓ_2 to ℓ_3 and the request $Path(n,\ell_2)$ can be replaced by the request $Path(\ell_2,\ell_3)$.

     – $Path(\ell_1,\ell_3)$ is requested. If $Path(n,\ell_2)$ can be replaced by $Path(\ell_2,\ell_3)$ we are done. Otherwise, when $Path(n,\ell_2)$ is requested, FIFO has already a path from ℓ_2 to ℓ_3 that does not pass through n. Then either $Path(\ell_1,\ell_3)$ was requested before $Path(n,\ell_2)$ and therefore $Path(n,\ell_2)$ does not produce a fault (a contradiction), or $Path(\ell_1,\ell_3)$ will be requested after $Path(n,\ell_2)$ but it will not produce a fault (a contradiction).

3. One simple path maximizes S. Suppose that some tree T of the forest F the adversary has in its cache is not a simple path. Let ℓ_h, ℓ_i and ℓ_j be three leaves of T such that the path from ℓ_h to ℓ_i is a longest path of T. We show that there exists a tree T' for which the value S is larger. T' is obtained from T by eliminating the edge that connects ℓ_j to T and adding an edge from ℓ_j to either ℓ_h or ℓ_i. By case 2 above we can assume that there are no queries $Path(n,\ell_j)$, but there could be one or more queries $Path(\ell,\ell_j)$, where ℓ is a leaf different from ℓ_h, ℓ_i and ℓ_j. Note that either $Path(\ell_h,\ell_j)$ or $Path(\ell_i,\ell_j)$ is not shorter than the path from ℓ to ℓ_j. Therefore we can eliminate $Path(\ell,\ell_j)$ replacing it with one of the other two queries.

   For a fixed ℓ_j, the adversary will never request both $Path(\ell_h,\ell_j)$ and $Path(\ell_i,\ell_j)$, because that would prevent him from asking $Path(\ell_h,\ell_i)$, that by hypothesis is the longest path of T. Therefore he will request only the longest of them, say $Path(\ell_h,\ell_j)$. But then T could be replaced with a different tree T' in which instead of ℓ_j there is a leaf $\ell_{j'}$ attached to ℓ_i, and all the queries $Path(n,\ell_i)$ or $Path(n,\ell_j)$ could be replaced by queries $Path(n,\ell_{j'})$ incrementing the overall length of the requested paths.

   Repeating this reasoning for T' we arrive to the conclusion that each connected component of F must be a path. But then, the best alternative for the adversary is to have only one path (of length k), for which he can ask one request of length k, two of length $k-1$, two of length $k-2$, and so on. This completes the proof of the theorem.

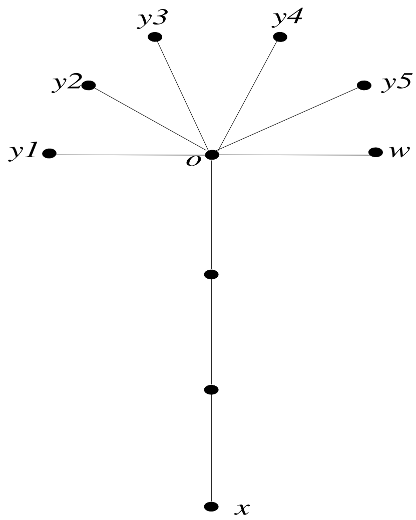

Figure 3: Example for Theorem 2.3 with $k = 8$, (o, w) is the edge requested at the beginning of the phase.

□

The proof of Theorem 2.1 is valid even assuming that the same path is provided to both the on-line algorithm and the adversary when they fault on the same query. If we eliminate this assumption, a stronger bound (that matches the upper bound of Theorem 2.2) can be obtained with a simpler proof by requesting paths

$$Path(v_1, v_{k+1}), Path(v_1, v_k), \ldots, Path(v_1, v_{\kappa+1})$$

and then

$$Path(v_2, v_{k+1}), Path(v_3, v_{k+1}), \ldots, Path(v_\kappa, v_{k+1}).$$

These requests could be served by the adversary with a path from v_1 to v_k, while if the on-line algorithm is provided with disjoint paths, he would need a cache of size at least $\lceil \frac{3}{4}k^2 \rceil$ to be competitive. Of course, for this we need to assume that the underlying graph contains all such disjoint paths.

2.2 Path-Paging for trees

In this section we show that if we restrict the input graph to be a tree then it is possible to improve the bounds given by Theorems 2.1 and 2.2. The improvement is due to the fact that, between each pair of vertices there is only one path; hence it is not necessary a cache of size quadratic with respect to the cache of the adversary for being able to "learn" the adversary's configuration.

2.2.1 Full-cost model

For this cost model, we first prove the following lower bound.

Theorem 2.3 *No deterministic on-line strategy with cache size k for the Path-paging problem in trees under the full-cost model is c-competitive with $c < \lfloor k^2/4 \rfloor + 1$, against an adversary with cache of size k.*

Proof: For the sake of simplicity, we will prove the theorem only in the case k is even; if k is odd the proof is similar.

Consider the following initial configuration of both the cache of the adversary and on-line algorithm A: there are $\frac{k}{2} - 1$ edges $\{(o, x_1), (x_1, x_2) \ldots (x_{\frac{k}{2}-2}, x_{\frac{k}{2}-1})\}$ forming a path from vertex o to vertex $x_{\frac{k}{2}-1} = x$, and $\frac{k}{2} + 1$ edges $(o, y_1) \ldots (o, y_{\frac{k}{2}+1})$. The adversary may then ask for a path (consisting of a single edge) between o and a vertex w such that $w \notin \{o, x_1 \ldots x_{\frac{k}{2}-1}, y_1 \ldots y_{\frac{k}{2}+1}\}$ (see figure 3).

Among all the edges in the initial configuration and the newly requested edge (o, w), there is at least one edge missing to A, and hence there exists a request $Path(x, z_1)$, $z_1 \in \{y_1 \dots y_{\frac{k}{2}+1}\} \cup \{w\}$ such that not all the edges necessary for its answer are present in A's cache. In order to answer this query the algorithm pays $k/2$ and must evict another edge; therefore there exists a request $Path(x, z_2)$, $z_2 \in \{y_1 \dots y_{\frac{k}{2}+1}\} \cup \{w\}$ such that not all the edges necessary for its answer are present in A's cache. In this way a sequence of $\frac{k}{2}$ queries $Path(x, z_i)$, $i = 1, 2, \dots, \frac{k}{2}$ are asked , making A pay $\frac{k}{2}$ (the length of paths (y, z_i)) at every query.

Queries $Path(x, w)$, $Path(x, z_i)$, $i = 1, 2, \dots, \frac{k}{2}$ form a phase; clearly A pays $k^2/4 + 1$ in order to answer all queries of the phase. However the adversary can pay only 1 during the phase; in fact when edge (o, w) enters in the cache the adversary can keep the path (o, x), and the edges (o, y_i) $i = 1, 2, \dots, \frac{k}{2}$ necessary to answer all queries $Path(x, z_i)$ $i = 1, 2, \dots, \frac{k}{2}$ with no cost, and hence the total cost for the adversary during the phase is 1 given by the cost for answering the first query of the phase.

At the end of the phase, a similar phase can start if any edge (o, z) neither in A's nor in the adversary's cache is requested, yielding an arbitrarily long sequence in which the cost of A is at least $k^2/4 + 1$ times the cost of the adversary. □

The next theorem shows that FIFO is at most a constant factor away from optimal for this problem.

Theorem 2.4 *FIFO with cache size k is $(k^2 + k)/2$-competitive for the Path-paging problem for trees under the full-cost model against an adversary with cache size k.*

Proof: Consider the sequence of queries divided in phases, as in the proof of Theorem 2.2. The paths requested during a phase involve at most k edges, and the cost of the adversary is at least 1.

Without loss of generality we assume that each query requires only one edge to be brought into the cache. In fact, suppose that in order to answer $Path(a, b)$ two or more edges must be brought into the cache.

Let $(a, x_1), (x_1, x_2) \dots, (y, y')(w_1, w_2)(w_2, w_3), \dots, (z, z'), (v_1, v_2), (v_2, v_3), \dots (v_i, b)$ be the requested path and let (y, y') and (z, z') be the missing edges; if we replace query $Path(a, b)$ with queries $Path(a, y')$ and $Path(a, b)$ then the total cost paid by FIFO is not smaller than the cost of answering only $Path(a, b)$; note that this can be done without affecting the cost of future requests, and hence increasing the total cost paid by FIFO to process the sequence.

Observe that if the phase consists of only one query then, obviously, the cost of FIFO is bounded by k; if there are two queries then the maximum cost is bounded by $(k - 1) + k$. In a similar way we have that if the phase consists of h queries then the cost of FIFO is bounded by $\sum_{i=k-h+1}^{k} i$; therefore the total cost during the phase is bounded by $\sum_{i=1}^{k} i$, that is $(k^2 + k)/2$. □

It is easy to see that the above competitiveness ratio is tight for FIFO. Assume that the configuration of FIFO consists in a k-edge path from vertex v_1 to vertex v_{k+1}, and such that the order in which the edges will be evicted is the reverse order of the indexes. This situation can be always forced by the adversary. Suppose that the next query is $Path(v_{k+1}, v_{k+2})$, and that the adversary evicts edge (v_1, v_2). The adversary may then ask for $Path(v_k, v_{k+2}), Path(v_{k-1}, v_{k+2}), \dots Path(v_2, v_{k+2})$. To make place for each new edge necessary to answer the query, FIFO will evict the edge necessary to answer the following one, and hence will have a fault at each query. The total cost for FIFO will hence be $(k^2 + k)/2$, while the adversary's is only 1. The configuration at the end of this sequence of requests is similar to the initial one (up to renaming of the vertices), and hence the pattern can be repeated forever.

If we consider randomized algorithms, it is possible to prove the following theorem.

Theorem 2.5 *Under the full-cost model and with a cache of size k, for sufficiently large k, no randomized on-line strategy for the Path-paging problem is better than $\Omega(k \ln k)$-competitive against an oblivious adversary with cache size k.*

Proof: The proof follows the approach developed in [9] of constructing a nemesis sequence for the on- line algorithm based on the possibility that an oblivious adversary has of knowing a probability distribution on the on-line's cache. The sequence σ is random, and we show that for each constant d, in order to have

$$E[C_A(\sigma)] - c * E[C_{ADV}(\sigma)] < d, \tag{1}$$

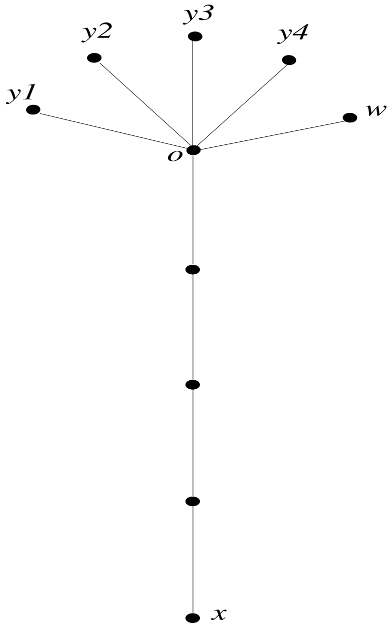

Figure 4: Example for Theorem 2.5 with $k = 8$, (o, w) is the edge requested at the beginning of the phase.

c must be $\Omega(k \ln k)$, where $C_A(\sigma)$ and $C_{ADV}(\sigma)$ are the random variables denoting the costs charged to algorithm A and to the adversary, respectively, for serving the sequence σ.

Assume the initial configuration in which both the adversary and A have in their caches the following set of edges: l edges $\{(o, x_1), (x_1, x_2) \dots (x_{l-1}, x_l)\}$ forming a path between vertex o and vertex x and $k - l$ edges $(o, y_1) \dots (o, y_{k-l})$ (the parameter l will be determined later). Consider a random sequence of queries composed by an arbitrarily large number of epochs; each epoch is defined as follows: the first query is $Path(o, w)$, $w \notin \{x_1 \dots x_l\} \cup \{y_1 \dots y_{k-l}\}$; in order to answer the query the adversary evicts edge (o, y_j), for some $j \in \{1 \dots k - l\}$ uniformly chosen at random.

The epoch proceeds with $k - l - 1$ subepochs, where each subepoch consists in zero or more requests to paths (x, y_i), $i \neq j$ already requested in the epoch followed by a request to a path $(x, y_{i'})$, $i' \neq j$ not yet requested in the epoch. The requests of the first type will be done as many times as necessary till the probability that the on-line algorithm has exactly the edges needed to answer all the requests so far in the epoch is 1. This is always possible whenever the on-line algorithm achieves a bounded competitive ratio as the adversary can simulate the behavior of the on-line algorithm on the sequence of requests up to that moment, and all these requests cost nothing to the adversary.

Upon completion of the epoch vertex w is "renamed" y_j so as to always have again the initial configuration (see figure 4).

The expected cost charged to the adversary in each epoch is 1 (in fact 1 is the exact cost). The cost of the algorithm is 1 for the first query of the epoch; in the full cost model each subsequent fault costs $(l + 1)$. Without loss of generality we may suppose that edges belonging to the path that connects o and x will not be evicted by the on-line algorithm and that l lines of the cache are filled by the edges of the path (o, x). In fact, if A ever evicts during the epoch an edge of the path (o, x), it will surely fault in order to answer the following query. Therefore, the expected number of faults on the rest of the epoch depends on the probability that edges (o, y_j) are in the remaining part of the cache, of size $k - l$. Equivalently this number is one less than the expected number of faults for an epoch of the Paging problem with cache of size $k - l$ (page y_i corresponds to path $Path(x, y_i)$). It has been proved in [9] (where epochs are called *phases*) that this number is greater than H_{k-l-1} (where H_x denotes the x-th harmonic number). Hence the expected cost for the epoch is greater than $1 + (l + 1)(H_{k-l-1} - 1)$.

Turning to inequality 1, it follows that c, the competitiveness coefficient, must be greater or equal to the maximum value of the previous expression. Simple calculations show that the maximum value is

$\Omega(k \ln k)$ and that, for $k \geq 4$, the value is greater than $k/2(\ln(k) - 2$. □

In the following we will prove that the Marking algorithm (Marking, for short) is nearly optimal under the full-cost model. Marking has been originally proposed in [9] for the paging problem; we extend it to the path paging problem as follows. The algorithm maintains a set of marked edges. Initially the marked edges are exactly those that are present in the cache. The marks are updated after each request: when query *Path*(a, b) is requested all edges of the path are marked. If the requested path is completely present in cache, then nothing else is done.

If edges in the cache do not allow to answer a query then there are two possibilities. Let x be the number of edges missing in the cache in order to complete the requested path. If there are at least x unmarked edges in the cache, then the algorithm evicts x edges that are randomly chosen among the unmarked ones to make place for the missing edges. If the number of unmarked edges is less than x then all marks except those assigned to edges of the current query are erased and then the previous case applies.

Before proving the competitive ratio of Marking we need some preliminary definitions. Consider the sequence of requests divided in epochs. During an epoch, requests may involve three different kinds of edges:

- *marked* edges, that are edges already used during the current epoch,
- *clean* edges, that are edges that where not used during the current epoch nor in the previous one,
- *stale* edges, that are edges that where used in the previous epoch but not during the current epoch.

In a similar way, we can divide the requests in four types:

- *clean* requests, that use clean and possibly marked edges,
- *stale* requests, that use stale and possibly marked edges,
- *mixed* requests, that use clean and stale edges, and possibly marked edges, and
- *marked* requests, that use only marked edges.

Without loss of generality we can suppose that there will not be requests of the last kind, as by definition Marking would answer them with no cost. It is easy to see that each epoch starts with a clean or a mixed request, that is, with a request that involves at least one clean edge.

Lemma 2.6 *The expected number of faults of Marking during an epoch is maximized if each query involves at most one clean or stale edge.*

Proof: Suppose a query q involves $r > 1$ stale or clean edges. We prove the lemma assuming that $r = 2$; by induction the proof can be easily extended to any other value of r. Let a and b denote the stale or clean edges of the query.

Let A (B) denote the event that edge a (b) is in the cache; $\overline{A}$ ($\overline{B}$) denotes the event that a (b) is not in the cache.

The expected number of faults F_q for answering q is equal to the probability that at least one edge among a and b is not present in the cache. Hence we can write:

$$F_q = \text{Prob}(\overline{A} \wedge \overline{B}) + \text{Prob}(A \wedge \overline{B}) + \text{Prob}(\overline{A} \wedge B).$$

Let q_a, q_b be two queries obtained from q in such a way that q_a (q_b) can be answered by including edge a and without edge b (respectively including b and not a) and let $\overline{B'}$ be the event that edge b is not in the cache *after* query q_a has been answered. The expected number F_{q_a,q_b} of faults for answering both queries q_a and q_b is equal to

$$F_{q_a,q_b} = \text{Prob}(\overline{A} \wedge \overline{B}) + \text{Prob}(\overline{A} \wedge B) + \text{Prob}(\overline{B'}).$$

The lemma follows by observing that $\text{Prob}(\overline{B'}) \geq \text{Prob}(\overline{A} \wedge \overline{B}) + \text{Prob}(A \wedge \overline{B})$ and, hence, $F_{q_a,q_b} \geq F_q$. □

Theorem 2.7 *Marking is $2kH_k$ competitive for the Path-paging problem on trees under the full-cost model, against an adversary with cache size k.*

Proof: By lemma 2.6 we can suppose that each request involves only one clean or stale edge, and possibly some marked edges.

This proof is similar to the proof of the competitiveness of Marking in [9]; for this reason we limit to give a sketch of the proof. The expected number of faults of Marking during the epoch is smaller than the number of faults it would have if all the clean requests of the epoch where done before any stale request. In such a case Marking has one fault for each clean request. Let c be the number of clean edges that are requested in the epoch, therefore, the number of faults for this type of requests is c. The probability of having a fault in a stale request is equal to the probability that the involved stale edge is not present in the cache. If we denote by s the current number of stale edges (that ranges from k to $c+1$), this probability is $\frac{c}{s}$. Hence, the expected number of faults of Marking during the epoch is less than

$$c + \frac{c}{k} + \frac{c}{k-1} + \ldots + \frac{c}{c+1} = c(1 + \frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{c+1}) = c(1 + H_k - H_c) \leq cH_k$$

As for the cost charged to the adversary, it can be proved in the same way as in [9] that it is at least $c/2$, half of the number of clean edges requested in the epoch.

It follows that the ratio between the expected number of faults of Marking and the adversary is equal to $2H_k$. In the full cost model, the maximum cost charged to Marking for a fault is k, while the minimum cost for a fault of the adversary is 1. Hence, if we consider the cost instead of the number of faults, we have that the cost of Marking is less than $2kH_k$ times the cost of the adversary, that is, Marking is $2kH_k$-competitive. □

The two previous theorems assert that Marking is at most a factor of 4 away from optimality.

2.2.2 Partial-cost model

In this section we consider the partial-cost model, in which the cost charged for a fault is the number of missing edges that complete the requested path.

In the following we propose a strongly competitive algorithm, based on the *Least Recently Used* (LRU) algorithm for the Paging Problem [18]. The LRU algorithm is defined as follows: when a fault occurs and new edges must be included in the cache LRU evicts the minimum number of edges that allows to include edges of the current query; the evicted edges are those that have not been used for the longest time (breaking ties arbitrarily).

The strategy can be implemented in the following way: edges are ordered in such a way that if e precedes e' then e' has not been used for answering a query after e has been used.

To answer a request $Path(a, b)$ the algorithm works as follows:

```
If a path P from a to b is present in cache
then
     A.1 Reorder edges in the cache by placing ahead edges
         belonging to P while maintaining their relative order
else
     B.1 Retrieve a path P from a to b.  Let x be the
         number of edges of P not present in the cache
     B.2 Reorder edges in the cache by placing ahead edges
         belonging to P that are already in the cache
         (maintaining their relative order) and then remaining
         edges (maintaining their relative order)
     B.3 Evict the last x edges in the cache that do not belong to P
     B.4 Introduce the x new edges of P placing them ahead in the cache.
```

Theorem 2.8 *LRU is k-competitive for the Path-paging problem on trees under the partial-cost model, against an adversary with cache size k.*

Proof: For every edge e of the tree let $a[e]$ be an integer valued variable, in the range $1, \ldots, k$ for all the edges present in the cache, such that the first edge in the cache (relative to the order maintained by LRU) has value k, the second one value $k-1$ and so on; if e is not in the cache then we assume $a[e] = 0$. $a[e]$ is the *weight* of e. Let us define the following potential function

$$\Phi = \sum_{e \in ADV} (k - a[e])$$

where ADV is the set of he adversary's cache.

We suppose that the adversary serves first each request, and then LRU serves it. The behavior of the potential function in the different situations is the following.

The adversary serves the request

Let z be the cost charged to the adversary (i.e. z new edges enter the adversary's cache). The maximum possible increase in the potential occurs when all the z new edges are not present in LRU's cache; in this case $\Delta\Phi \leq kz$.

LRU serves a request

If LRU does not fault then there is no increase in the potential function; in fact there might be edges in both LRU and adversary's cache that decrease their weight; however this decrease is equal or greater to the increase of the weights for edges that allow to answer the query (and that certainly belong to the adversary's cache).

If LRU faults then let y be the length of the path from a to b, x is the number of edges of that path that are not present in LRU's cache. The modifications that LRU performs on the weights of the edges can be described as follows:

B.2 Assign to the edges of the cache that are not in the path from a to b the values $1, \ldots, k-(y-x)$ maintaining their relative order; assign values in the range $k-(y-x)+1, \ldots, k$ to the edges of the cache that are in the path;

B.3 for every edge e of the cache do $a[e] := \max(a[e] - x, 0)$; evict edges of the cache with weight equal to 0;

B.4 bring into the cache the missing edges, arbitrarily assigning weights in the range $k-x+1, \ldots, k$.

Step B.2 above increases the weight of $(y-x)$ edges in the adversary's cache and this increase is at least the decrease of edges in the adversary's cache that are not part of the query path; therefore there is no increase in the potential due to this step. Since there are no more than $(k-x)$ edges in the intersection of both caches before the execution of step B.3, then the increase of the potential function due to step B.3 is at most $(k-x)x$. Since the x new edges are placed ahead of LRU's cache, then the change of the potential function due to the insertion of the new edges (step B.4) is $-(kx - \sum_{i=1}^{x-1} i)$. Therefore the total modification of the potential function is at bounded by

$$\Delta\Phi \leq kx - x^2 - kx + \sum_{i=1}^{x-1} i = -x^2 + \frac{(x-1)x}{2} \leq -x$$

Lemma 1.1 implies that LRU is k-competitive. □

In the sequel of this section we will show that the Marking algorithm achieves a competitive ratio of $2H_k$ by considering a particular kind of adversary. Let $d(a,b)$ be the number of edges in the path connecting a and b.

The restricted adversary, before every query $Path(a,b)$ with $d(a,b) > 1$(from now on these kind of queries will be referred to as *long* queries), must request all the edges that form the path from a to b, in any order; let $\mathcal{P}(a,b)$ be such a sequence. This is not a real restriction; in fact the cost charged to the restricted adversary is exactly the same that it would be charged to a general adversary. Furthermore the cost charged to Marking is at least the same it would have to pay in a general sequence; in fact, in the partial-cost model, an algorithm is charged exactly the number of missing edges; hence the cost of Marking for answering all queries in $\mathcal{P}(a,b)$ and then $Path(a,b)$ is at least the number of missing edges in $Path(a,b)$ before the first edge of $\mathcal{P}(a,b)$ is requested.

Moreover, we assume a *lazy* adversary, (i.e. an adversary that does not evict any edge if all edges necessary to answer the current query are in the cache, and, when a fault occurs, evicts exactly the number of edges necessary to make place for the missing ones). It has been shown in [15] that this latter assumption can be done without loss of generality.

As in the previous section, edges may be divided in marked, clean or stale, and requests in clean, stale, mixed and marked. The proof of the following lemma is trivial and it is omitted.

Lemma 2.9 *Each epoch starts with a clean request.*

Lemma 2.10 *Every long query but the first one of an epoch is served by Marking with no cost.*

Proof: By definition of Marking and the kind of adversary we are considering, all edges that allow to answer a long query $Path(a, b)$ but the first one of an epoch have been already marked and, hence, are in the cache. □

Theorem 2.11 *Marking is $2H_k$ competitive for the Path-paging problem under the partial-cost model, against an adversary with cache size k.*

Proof: By lemma 2.10 we can eliminate from the input sequence all long queries except the first one of each epoch; furthermore, since Marking does not pay for marked edges then we can eliminate from the input sequence all marked queries. Therefore we can assume that the sequence of requests of an epoch is as follows:

$$c_1, \ldots, c_i, sc_1, \ldots, sc_j, L, c_1^1, c_2^1, \ldots, c_{x_1}^1, s_1^1, s_2^1, \ldots, s_{y_1}^1, c_1^2, c_2^2, \ldots, c_{x_2}^2, s_1^2, s_2^2, \ldots, s_{y_2}^2, \ldots$$

where $c_1 \ldots c_i$ are requests to clean edges, $sc_1 \ldots sc_j$ are requests to clean or stale edges, L is either empty or a long query involving edges $c_1, \ldots, c_i, sc_1, \ldots, sc_j$ and possibly some other stale edges (denoted $s', s'', s''', \ldots$ in the sequel) that have been requested in the final part of the previous epoch. Queries c_m^n and s_m^n are clean and stale requests for one-edge paths, respectively.

Since a marked edge is not evicted it follows that the cost charged to Marking for this sequence is less that the cost it would be charged for the sequence

$$c_1, \ldots, c_i, sc_1, \ldots, sc_j, s', s'', s''', \ldots, c_1^1, c_2^1, \ldots, c_{x_1}^1, s_1^1, s_2^1, \ldots, s_{y_1}^1, c_1^2, \ldots, c_{x_2}^2, s_1^2, \ldots, s_{y_2}^2, \ldots$$

The rest of this proof is similar to the proof of the competitiveness of Marking for the full-cost model. The expected cost charged to Marking for the above input sequence is smaller than the cost it would be charged if all the clean edges of the sequence where requested before any request to a stale edge. In such a sequence Marking pays one for each clean edge and an expected cost for each stale request that is equal to the probability that the edge is not present in the cache. This probability is $\frac{c}{s}$, where c is the number of clean edges requested so far and s is the current number of stale edges. Hence, the expected cost charged to Marking during the epoch is less than

$$c + \frac{c}{k} + \frac{c}{k-1} + \ldots + \frac{c}{c+1} = c(1 + \frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{c+1}) = c(1 + H_k - H_c) \leq cH_k$$

Following [9] it is possible to prove that the cost charged to the adversary is at least $c/2$, half of the number of clean edges requested in the epoch. This completes the proof of the $2H_k$-competitiveness of Marking.

□

We recall that in [9] it has been proved that H_k is a lower bound for the competitive ratio of any algorithm for the Paging problem, and hence it is also a lower bound for the Path paging problem for trees under this cost model. Therefore the proposed algorithm is at most a factor of 2 from optimality.

2.2.3 0/1-Cost Model

If the 0/1-cost model is used, we can prove that LRU is k-competitive and, hence, optimal.

Theorem 2.12 *LRU and FIFO are k-competitive for the Path-paging problem on trees under the 0/1 cost model, against an adversary with cache size k.*

Proof:
We prove the theorem for LRU omitting the analogous proof for FIFO. We will suppose that the adversary serves each request before LRU. An integer valued variable, in the range $1, \ldots, k$ is associated to all edges present in the cache in the same way that was done in the proof of Theorem 2.8; if e is not in the cache then we assume $a[e] = 0$.

Let us consider the following potential function

$$\Phi = \max_{e \in L, e \notin ADV} a[e]$$

where L and ADV denote, respectively, the set of edges in LRU's cache and in the adversary's cache. The behavior of the potential function is as follows.

The adversary answers the query

Since $0 \leq \Phi \leq k$ then clearly $\Delta\Phi \leq k$.

LRU answers the query

Let us first consider the case when LRU faults. Since the adversary serves each request first then the edges that enter into LRU's cache are all present in the adversary's cache. Moreover, since LRU faults there must be at least one edge e such that $e \in L, e \notin ADV$, then the update of the array a, produces a decrease in the potential function, i.e. $\Delta\Phi \leq -1$.

On the other side, if LRU serves a request without a fault then $\Delta\Phi \leq 0$.

The theorem follows by applying Lemma 1.1. □

Since k is a lower bound for the competitiveness of any deterministic algorithm it follows that LRU and FIFO are optimal. The following theorem provides a lower bound for randomized algorithms.

Theorem 2.13 *No randomized on-line strategy for the Path-paging problem on trees under the 0/1 cost model is c competitive against an adversary with cache size k with $c < (\lfloor \frac{k}{e} \rfloor + 1)$ (e is the base of the natural logarithm) even if oblivious adversaries are considered.*

Proof: Let A be any on-line algorithm; assume that before the first query is disclosed the adversary and A have the same set of edges in their caches. The input sequence is random and consists of an arbitrarily large number of rounds. The first request of each round asks a path of length l (the value of l will be determined later) that is disjoint from the set of edges present in both caches. To serve this request, the adversary evicts l randomly chosen edges; the round continues with $k - l$ subrounds; each subround is defined as follows: first, all requests of previous subrounds of the round are repeated as shown in the proof of Theorem 2.5; then one of the $k - l$ edges not evicted by the adversary is requested as one-edge path. The round terminates when all those $k - l$ edges have been requested.

The adversary's cost during the round is 1, and after a round A's and the adversary's cache coincide and a new round can start. To complete the proof it is sufficient to show that the expected cost charged to A during a round is at least $\frac{k}{e} + 1$.

A pays 1 for the first request of the round. The expected cost paid by the algorithm for answering the queries of a subround is greater than the probability that, for each of the $k - l$ subrounds, the requested edge is in the cache when it is requested for the first time during the round. The probability that the first edge that is requested during the i-th subround is in A's cache is $\frac{l}{k-i+1}$. Therefore the expected cost for a round is at least:

$$1 + \frac{l}{k} + \frac{l}{k-1} + \ldots + \frac{l}{l+1} = 1 + l(\frac{1}{k} + \frac{1}{k-1} + \ldots + \frac{1}{l+1}) = 1 + l(H_k - H_l)$$

The maximum value of the above expression is $1 + \frac{k}{e}$ (when $l = \frac{k}{e}$). This completes the proof of the theorem. □

Recall that deterministic algorithms LRU and FIFO are k competitive in this case.

3 Connectivity-paging

In this section we consider the problem of answering (in an on-line way) queries about the connectivity of an arbitrary directed graph. We assume that the cache contains a subset of k pairs of vertices, together with the information whether the two vertices lie in the same connected component of the graph or not. In other words, the cache is a subset of edges of the transitive closure of the graph (which we suppose is stored in secondary storage). A pair (i, j) is called a *yes-edge* if i and j are in the same connected component, a *no-edge* otherwise. Both kinds of pairs will be referred to as *edges* when no confusion arises. We assume that no further information is stored in the cache.

A query *Connected*(i, j) may be answered with no cost if there is a path in the cache containing at most one no-edge between i and j: if there is a path containing no no-edges, the answer to *Connected*(i, j) will be yes (pair (i, j) is called a *deduction*); conversely if there is a path containing one no-edge, the answer to the query will be negative (in this case we call pair (i, j) a *no-deduction*). Paths with more than one no-edge do not provide information to answer the connectivity query between i and j.

If the cache does not contain a path between i and j with at most one no-edge then the query must be answered by accessing secondary memory. We require that after each query *Connected*(i, j), the information about the connectivity between i and j must be present in the cache; namely, the faulty query *Connected*(i, j) is answered by bringing into the cache edge (i, j): neither the algorithm nor the adversary are allowed to bring into the cache any other edge of the transitive closure. We assume that a constant cost is charged for each fault.

3.1 Lower bounds

Let $f(k)$ be the maximum number of no-deductions that are consistent with a cache of size k. The value of $f(k)$ is crucial in our analysis. In fact, we will show that no strategy can achieve a bounded competitiveness coefficient against an adversary with a cache of size k if its own cache is of size $K < f(k)$; we will see that $f(k) \simeq k^2/2$.

Theorem 3.1 *No deterministic on-line strategy for the Connectivity-paging problem on arbitrary graphs can be competitive against an adversary with cache size k if its own cache is of size $K < f(k)$.*

Proof: Consider the sequence of $(K + 1)k$ requests $e_{11} \dots e_{1k} \dots e_{(K+1)1} \dots e_{(K+1)k}$ such that each subsequence $e_{i1} \dots e_{ik}$ denotes a set of yes- and no-edges that allows exactly $f(k)$ no-deductions. Since the memory of the on-line algorithm is K, after such a sequence there exists at least one i such that the on-line strategy has none of $e_{i1} \dots e_{ik}$ in its cache. This holds whichever were the initial configurations of the on-line strategy and that of the adversary.

Since the adversary knows the exact value of i, he can serve the sequence of request as follows: the subsequence $e_{11} \dots e_{1k} \dots e_{(i-1)1} \dots e_{(i-1)k}$ can be served in any way. For $e_{i1} \dots e_{ik}$ the adversary empties the cache to keep all the edges of the subsequence. For the remaining queries of the sequence, he pays for every query, caching all the edges to the same position of his cache, a position that contained some no-edge, say (x, y). After the last query of the sequence the adversary may ask again for *Connected*(x, y); after this query the adversary is able to answer a set $f(k)$ different negative queries with no cost. Note that before answering *Connected*(x, y) the on-line algorithm has no edge of this set; since $K < f(k)$, then there is always at least one no-edge absent from his cache. Therefore the algorithm has to pay for every query of an arbitrarily long sequence of queries for which the adversary pays nothing, yielding an unbounded relation between the costs. □

Lemma 3.2 *Given a cache of size k the maximum number of no-deductions is*

$$f(k) = \frac{k^2}{2} - 3\left(\frac{k}{2}\right)^{\frac{5}{3}} + O(k^{\frac{4}{3}}).$$

This value is obtained when the yes-edges of the cache form l, $l = \frac{k}{2}^{\frac{1}{3}} + \frac{3}{4} + O(k^{-\frac{1}{3}})$, spanning trees, whose sizes differ at most in one unit, and there is a no-edge in the cache between each pair of spanning trees (see figure 5).

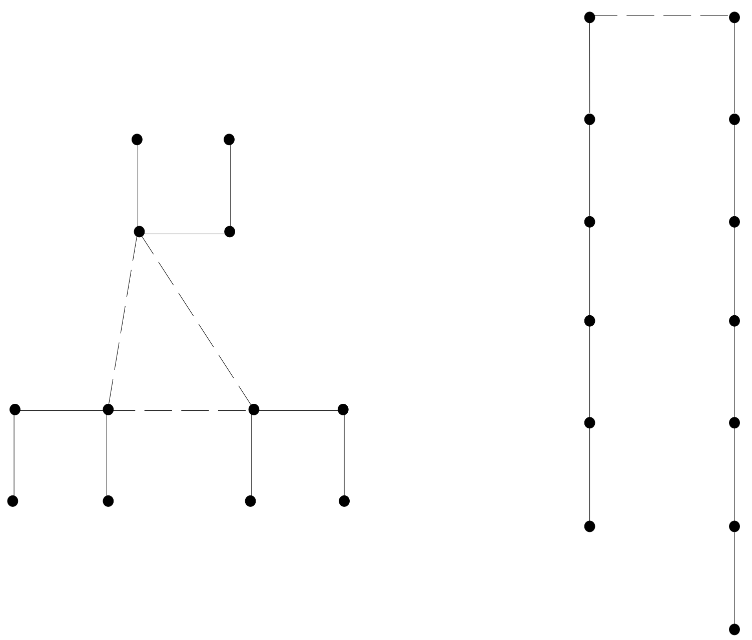

Figure 5: Two possible configurations for $k = 12$. The first one maximizes the number of possible no-deductions. Full lines represent yes- edges of the graph, while dashed lines represent no-edges.

Proof: We suppose that the underlying graph contains subgraphs of the desired type. Let CC be a connected component. Given an initial configuration of the cache with any number of connected components we first obtain a sequence of configurations with non-decreasing number of no-deduction that has fixpoints in configurations of the claimed type, but possibly with a different number of connected components.

The sequence is obtained by applying one of the following rules until the fixpoint is obtained. Without loss of generality we assume that at the beginning the initial configuration contains at least one no-edge (otherwise the number of no-deductions is zero).

Transformation Rules

1. If there exists two CCs a and b such that there is not a no-edge between a vertex in a and a vertex in b, then merge a and b identifying one vertex x of a and one vertex y of b in a new vertex z. If this results in repeating the same no-edge (z, w), then replace one copy of the edge by a positive edge (z, v) where v is a new vertex.

2. If any CC is not a spanning tree then eliminate one positive edge that belongs to a cycle and add a positive edge between a vertex in the CC and a new vertex.

3. If there are two no-edges between the same pair of CCs then eliminate one no-edge and add a new yes-edge (x, y) where x is a vertex of one CC and y is a new vertex that was not previously in any CC.

4. If the difference of the number of positive edges belonging to a pair of CCs connected by a no-edge is greater than one then eliminate a positive edge from the bigger CC and add a positive edge to the smaller one.

It is easy to see that rules 2,3 and 4 do not increase the number of possible no-deductions. As far as rule 1 is concerned we observe that all no-deductions that involve x and y are now possible with vertex z; moreover it might happen that new no-deductions become possible.

When no rule can be applied, the configuration consists in a set of CCs whose number of edges (and vertices) differ in at most one. Notice that the number of CCs obtained at the fixpoint depends on the order in which rules are applied.

It remains to find the optimal size of the CCs as a function of k. It is sufficient to maximize the number of no-deductions possible as a function of the number of connected components varies. For example, Figure 5 shows two possible configurations for $k = 12$, one with 3 CCs and one with 2 CCs.

If there are l CCs then the number of no-edges in such a configuration will be $\binom{\frac{l(l-1)}{2}}{}$; each CC contains either $(\lfloor\frac{k-\frac{l(l-1)}{2}}{l}\rfloor + 1)$ or $(\lfloor\frac{k-\frac{l(l-1)}{2}}{l}\rfloor + 2)$ vertices. The maximum number of no-deductions possible with a cache of size k is

$$f(k) = \max_l \left(\frac{k - \frac{l(l-1)}{2}}{l} + 1\right)^2 \frac{l(l-1)}{2} = \frac{k^2}{2} - 3\frac{k^{\frac{5}{3}}}{2} + O(k^{\frac{4}{3}}) \qquad (2)$$

The value of l that maximizes the value of $f(k)$ is $l \approx \frac{k}{2}^{\frac{1}{3}} + \frac{3}{4} + o(1)$.

□

The second result of this section is a lower-bound on the competitive ratio of any on-line algorithm, with a cache of any size.

Theorem 3.3 *No on-line algorithm for the Connectivity paging problem for arbitrary graphs with any cache size can be c-competitive against an adversary with cache size k with $c < k$.*

Proof: Let h be the size of the algorithm's cache; we assume that the underlying graph has at least h^{2+k} vertices. Consider a configuration in which the adversary has $k-1$ yes- edges forming a tree with vertex set T and the on-line algorithm has no edges joining vertices of T; this is always possible using a suitable input sequence of queries analogously to the sequence exploited in the proof of Theorem 3.1.

The rest of the input sequence is defined as follows: the adversary asks for a no-edge (x, j) where j belongs to the tree and x is neither in the adversary's cache nor in the algorithm's cache. After this query, k queries *Connected*$(x, t), t \in T$ can be answered by the adversary with no cost. Since the on-line algorithm has no yes-edges connecting vertices in T, it will have a fault for each one of these k queries.

Afterwards the input sequence continues analogously with query (y, j') where j' belongs to the tree and y is neither in the adversary's cache nor in the algorithm's cache; this pattern can be repeated an arbitrary number of times. The lower bound follows by observing that for each of these subsequences 1 and k are, respectively, the adversary's and the algorithm's costs. □

3.2 An upper bound

In the following, we will prove an upper bound for the competitive ratio achieved by a variant of FIFO with a cache of size $f(k) + k$ for the Connectivity-paging problem.

The proposed algorithm maintains yes- and no-edges in separate parts of the cache, using the First-in-first-out policy in each of them. Namely, the algorithm (that we call FIFO^D) has at every moment k positive edges and $f(k)$ negative edges.

We consider the following implementation of FIFO^D: a value $p(e)$ from 1 to k is associated with each positive edge e in the cache and a value $n(e)$ from 1 to $f(k)$ is associated with each negative edge in the cache. Whenever FIFO^D has a fault on a positive (negative) edge, the values of $p(e)$ $(n(e))$ are decreased for every e such that $p(e) > 0$ $(n(e) > 0)$. The edge e such that $p(e)$ $(n(e))$ becomes exactly 0 is the one evicted, and the arriving edge is given value k $(f(k))$.

Let $D(ADV)$ denote the set of pairs of vertices such that the adversary may answer without paying a query about the connectivity of that pair in a certain configuration ADV of its cache.

Proposition 3.4 *Whenever FIFO^D has a fault on a positive (negative) edge, there must be at least one edge e with $p(e) > 0$ $(n(e) > 0)$ such that $e \notin D(ADV)$.*

Theorem 3.5 *FIFO^D with a cache of size $f(k)+k$ is $(f(k)+k)$-competitive for the Connectivity-paging problem on arbitrary graphs against an adversary with cache size k.*

Proof: The following potential function is used:

$$\Phi = \max_{e \notin D(ADV)} \{p(e)\} + \max_{e \notin D(ADV)} \{n(e)\}$$

We will suppose that the adversary serves each request first, and then it presents it to FIFO^D. Again, we will analyze the behavior of the potential function in the cases where the adversary and FIFO^D move.

The adversary moves, evicting edge (i, j)

Since $\Phi \leq k + f(k)$, obviously $\Delta\Phi \leq k + f(k)$

FIFO^D faults on request *Connected*(i, j)

Note that in this case the entering edge (i, j) is deducible by the adversary (as the adversary serves each request first). The decrement in the value of p (n) for all the positive (negative) edges of FIFO^D produces a decrement in the potential function whenever there was at least one edge e with $p(e) \geq 0$ ($n(e) \geq 0$), $e \notin D(ADV)$. By proposition 3.4 this is always the case, and hence $\Delta\Phi \leq -1$.

By Lemma 1.1, FIFO^D is $f(k) + k$-competitive. □

4 Link paging

An important feature of high speed networks (e.g. ATM networks) is that the cost of establishing a communication channel between two vertices of the network depends on the reconfiguration of the network that is eventually necessary in order to establish the channel. Namely we assume a computer network in which all possible connections between pairs of vertices might occur, but at most k pairs of vertices are connected at the same instant. Communication requests are pairs of vertices to be connected, and one request is served with no cost if there is a path between the vertices in the current configuration of the k links; otherwise, both vertices are linked by switching one of the k physical links to connect the requested pair, and the cost is 1. We call this problem the *Link-paging* problem. An algorithm for the Connectivity-paging problem in the particular case in which the graph is complete is also an algorithm for this problem.

In the following we will prove that FIFO with a cache of size K is $\frac{K}{K-k+1}$-competitive against adversaries that use caches of size $k \leq K$, and therefore optimal for the link paging problem.

Theorem 4.1 *FIFO with cache size K is $\frac{K}{K-k+1}$-competitive for the Link-paging problem against adversaries with cache size $k \leq K$, and this is optimal.*

Proof: An integer valued variable, in the range $1, \ldots, K$ is associated with all edges present in the cache in the same way that was done in the proof of Theorem 2.8; if e is not in the cache then we assume $a[e] = 0$. Whenever FIFO has a fault $a[e]$ is decreased for every edge in the cache and the evicted edge is the edge e such that $a[e]$ becomes 0. The edge that is inserted into the cache in order to answer the query receives the value K.

Note that edges in FIFO's cache do not form a cycle; moreover without loss of generality we can assume that also the adversary's cache does not contain a cycle (since if this occurs the cache contains at least one redundant edge).

Let $G_{FIFO} = (V_{FIFO}, E_{FIFO})$ and $G_{ADV} = (V_{ADV}, E_{ADV})$ be the graphs induced respectively by the set of edges in FIFO's and the adversary's caches. Let H be the multigraph union of G_{FIFO} and G_{ADV} (where edges in both caches appear twice). Let C be the set defined by the following procedure:

```
C := ∅;
while there is a cycle in H do
   begin
   let e be the edge in FIFO's cache belonging to a cycle
   such that a[e] is minimum;
   C := C ∪ {e};
   remove the edge e from H
   end
```

C can be seen as the minimum-weight "feedback" edge set of H that consists only of edges in E_{FIFO}. We consider the following potential function:

$$\Phi = kK - \sum_{e \in C} a[e].$$

When the adversary faults, $\Delta\Phi \leq K$. This is true because the entering edge does not increase the potential function and the evicted edge can increase it by at most K. In order to show this latter fact, let $e = (a, b)$ be the evicted edge, and suppose that e was part of x cycles of G. If $x = 1$, then the eviction of e eliminates only that cycle and hence $\Delta\Phi \leq K$. If $x > 1$, then let $C_1 \ldots C_x$ be those cycles, and suppose without loss of generality that C_1 is the cycle whose minimum edge is maximum among the minimum edges of all the cycles. Then C_1 is the last cycle that is eliminated by the above procedure. But then, each one of $C_2 \ldots C_x$ is a cycle with e substituted by $C_1 - e$, so the edges considered in the computation of the new potential function will be the same as before the eviction of e, except the one corresponding to C_1.

The other thing to note is that if FIFO has a fault, then $\Delta\Phi \leq -K + k - 1$. To see this, note that the entering edge is necessarily part of a cycle with only edges that are in the adversary's cache, and, hence, it decreases the potential by K. Besides, note that $|C| \leq k$, because each cycle contains at least one edge of E_{ADV}, and neither the adversary nor FIFO have redundant connections. Moreover if $|C| = k$ then it follows that $D(ADV) \subseteq D(FIFO)$, where $D(ADV)$ and $D(FIFO)$ denote the set of pairs that may be connected by the adversary and FIFO respectively. Therefore, the decrease of the weights of the other edges in FIFO's cache produces an increase of the potential that is at most $k - 1$; in fact if FIFO faults, there are at most $k - 1$ edges in C. Hence $\Delta\Phi \leq -K + k - 1$.

By applying Lemma 2.1 the thesis follows. □

As in the case of the general connectivity problem, it is interesting to note that LRU will not work for this problem.

5 Conclusions and open problems

In this paper we have studied two extensions of the Paging problem to graphs. We have shown that the competitiveness results that hold for the Paging problem become much worse if we assume that query answers are either a set of items that are constrained to form a path in an underlying graph or the connectivity information about a pair of vertices.

Many open problems require further investigation. First of all it would be interesting to close the gaps between upper and lower bounds that do not match.

Furthermore it would be interesting to extend Path-paging to directed and/or non-connected graphs, and to consider particular classes of graphs such as trees, forests or graphs with bounded diameter etc.

A topic that deserves further study arises by noting that the lower bound of theorem 2.1 does not hold if we consider a different model in which at every fault on query *Path*(a, b) the on-line algorithm may bring into cache any set of edges and not necessarily a shortest path from a to b. A similar extension of the Connectivity-paging problem consists in considering algorithms that in response to a fault on query *Connected*(a, b) are not forced to bring to their cache the edge (a, b) of the transitive closure but may look for other edges that allow to answer the query. Note that the proof of the lower bound on the size of the cache in theorem 3.1 does not hold in this model.

Acknowledgments: We would like to thank Giorgio Ausiello for useful discussions about this work, and Ricardo Baeza-Yates for computing the exact value of $f(k)$ in Lemma 3.2. We also acknowledge anonymous referees for comments and observations that allowed us to improve the presentation.

References

[1] A. Aggarwal, B. Alpern, A.K. Chandra and M. Snir, A model for hierarchical memory, *Proc. 19th Annual ACM Symposium on Theory of Computing* 305-314 (1987).

[2] A. Aggarwal and A.K. Chandra, Virtual memory algorithms, *Proc. 20th Annual ACM Symposium on Theory of Computing* 173-185 (1988).

[3] A. Aggarwal, A.K. Chandra and M. Snir, Hierarchical memory with block transfer, *Proc. 28th. Annual Symposium on Foundations of Computer Science* 204-216 (1987).

[4] J.L. Bentley and C.C. McGeoch, Amortized analysis of self organizing sequential search heuristics, *Comm. ACM* **28(4)** 404-411 (1985).

[5] A. Borodin, Sandy Irani, P. Raghavan and B. Schieber, Competitive paging with locality of reference, *Proc. 23rd Annual ACM Symposium on Theory of Computing* 249-259 (1991).

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

[7] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan, New Results on server problems, *Proc. 1st. Annual ACM-SIAM Symposium on Discrete Algorithms* 291-300 (1990).

[8] M. Chrobak and L. Larmore, An optimal online algorithm for k servers on trees, *SIAM Journal on Computing* **20** 144- 148 (1991).

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

[10] A. Fiat, Y. Rabani and Y. Ravid, Competitive k- server algorithms, *Proc. 31st. Annual Symposium on Foundations of Computer Science* 454-463 (1990).

[11] E.F. Grove, The harmonic online k-server algorithm is competitive, *Proc. 23rd Annual ACM Symposium on Theory of Computing*, 260-266, (1991).

[12] A. Karlin, M. Manasse, L. Rudolph and D. Sleator, Competitive snoopy caching, *Algorithmica* **3** 79-119 (1988).

[13] E. Koutsoupias, C. Papadimitriou, On the k-server conjecture, *Journal of the ACM* **42(5)**, 971-983 (1995).

[14] M. H. Nodine, M. T. Goodrich and J. S. Vitter, Blocking for external Graph Searching, *Algorithmica* **16(2)**, 181-214 (1996).

[15] M.S. Manasse, L.A. McGeoch and D.D. Sleator, Competitive algorithms for server problems, *Journal of Algorithms* **11(2)**, 208-230 (1990).

[16] M. Overmars, M. Smid, M. de Berg and M. van Kreveld, Maintaining range trees in secondary memory, part I: partitions, *Acta Informatica* **27** 423-452 (1990).

[17] P. Raghavan and M. Snir. Memory versus randomization in on-line algorithms, *IBM Journal of Research and Development*, 38(6):683–707, November 1994.

[18] D. Sleator and R.E. Tarjan, Amortized efficiency of list update and paging algorithms, *Comm. ACM* **28** 202-208 (1985).

[19] M. Smid and P. van Emde Boas, Dynamic data structures on multiple storage media, a tutorial, *Technical Report Universitat des Saarlandes* **a 15/90**, (1990).

[20] N. Young, On-line caching as cache size varies, *Proc. 2nd. Annual ACM-SIAM Symposium on Discrete Algorithms* 241-250 (1991).