ONLINE ALGORITHMS FOR LOCATING CHECKPOINTS

by

Marshall Bern, Daniel H. Greene, Arvind Raghunathan,
and Madhu Sudan

Memorandum No. UCB/ERL/IGCT M91/17

1 March 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

ONLINE ALGORITHMS FOR LOCATING CHECKPOINTS

by

Marshall Bern, Daniel H. Greene, Arvind Raghunathan,
and Madhu Sudan

Memorandum No. UCB/ERL/IGCT M91/17

1 March 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Online Algorithms for Locating Checkpoints

Marshall Bern [*] Daniel H. Greene [*] Arvind Raghunathan [†]
Madhu Sudan [‡]

Abstract

Motivated by applications in data compression, debugging, and physical simulation, we consider the problem of adaptively choosing locations in a long computation at which to save intermediate results. Such checkpoints allow faster recomputation of arbitrary requested points within the computation. We abstract the problem to a server problem in which k servers move along a line in a single direction, modeling the fact that most computations are not reversible. Since checkpoints may be arbitrarily copied, we allow a server to jump to any location currently occupied by another server. We present online algorithms and analyze their competitiveness. We give lower bounds on the competitiveness of any online algorithm and show that our algorithms achieve these bounds within relatively small factors.

1. Introduction

Suppose you are building software for accessing an encyclopedia. To save space, you store the encyclopedia in compressed form using an adaptive data compressor [8, 15]. Your software must handle requests from users wishing to read arbitrarily-located articles within the encyclopedia. Here a problem arises: in order to decompress a specific article, you must recreate the compression statistics as they were at the time that article was compressed.

There are several possible approaches to this problem. One could save all compression statistics, but this defeats the purpose of compression. One could break the encyclopedia into smaller files—or equivalently restart the compressor occasionally—but this, too, compromises compression. A similar solution is to occasionally save—or checkpoint—the compression statistics while compressing; then a request is handled by finding the closest previous checkpoint and recomputing statistics from that point up to the requested article. The space cost of a checkpoint is typically comparable to (perhaps twice) the cost of restarting the compressor. The most flexible solution allows the locations of the checkpoints to move and adapt to the pattern of requests. In this paper we investigate adaptive solutions to the problem of locating checkpoints.

Besides data compression our work applies to a number of other contexts.

[*]Xerox PARC, 3333 Coyote Hill Rd., Palo Alto, CA, 94304
[†]Courant Institute, New York University
[‡]Computer Science Division, UC - Berkeley, supported in part by NSF grant CCR-8947792

- In debugging a long program, one typically probes an irreversible computation at various points in order to check intermediate values [10].
- In studying an irreversible physical system, one would like to interactively probe a computer simulation.
- In testing a VLSI design, different members of a design team may work on different parts of a critical path simultaneously. Thus a useful feature of a waveform simulator such as *Spice* would be the capability to answer probes at arbitrary points along a path. As above, the computation of a waveform is typically irreversible.

We model the problem as follows. (Here we use the terminology of the data compression application.) We can afford k "permanent" checkpoints; in addition, we set aside scratch space for one temporary checkpoint. We think of the temporary checkpoint as residing in fast memory so that it can be rapidly updated as we read through the encyclopedia; the other k checkpoints may reside in slow memory.

We are presented with a sequence of n requests, each at a real number in the half-open interval $[0, m)$. A permanent checkpoint may (1) move forward (towards larger numbers) along positions in $[0, m)$, incurring cost equal to the difference between starting and ending positions; (2) *fork*, that is, immediately move at no cost, to a position currently occupied by another checkpoint; or (3) *restart*, at no cost, at position 0. Any number of these moves may be made in response to a request. After these moves, the request at $r \in [0, m)$ is *serviced* by the temporary checkpoint, incurring fixed cost 1 plus the distance to the request from the closest checkpoint at a position no greater than r. In the terminology of Manasse et al [11], each request is serviced by an *excursion* from the nearest permanent checkpoint. The temporary checkpoint does not persist between requests.

We would like to minimize the total cost of a sequence of requests; that is, we are interested in maximizing throughput rather than minimizing worst-case latency. In our model, the only costs are computation; copying one block of memory to another, as in a fork move, is free.

For simplicity, we carry out our analysis assuming that position m coincides with position 0, that is, the encyclopedia is circular. This assumption eliminates move (3) and clarifies our arguments. We then show how to transfer our results back to the linear case.

We analyze the *competitiveness* of our algorithms [3, 9, 11, 14]. That is, we compare the performance of an online algorithm against the performance of an optimal offline algorithm that sees all requests in advance. An algorithm is called c-competitive if its cost on any sequence of n requests is at most $O(1)$ greater than c times the offline algorithm's cost. This style of analysis refines traditional worst-case analysis. Competitive analysis is worst-case in that no assumptions about the distribution or correlation of requests are made; however, it measures performance relative to what is achievable by an omniscient algorithm, rather than in absolute terms.

A discretized version of our problem is an example of a *task system* as defined by Borodin et al [3]. Borodin et al, however, study a more general model in which the costs of serving requests—rather than just request locations—may be chosen by an adversary, so their bounds have no nontrivial implications for our problem. Other related work includes a number of recent papers on *server problems* [2, 4, 5, 6, 7, 11, 13]. The paper that addresses

$(3/2)km$. The work done by the player is at most m, and the increase in Φ can be at most km, so the left-hand side is smaller than the right-hand side for $k \geq 2$.

Now assume that there is an adversary server j such that $d(x_j, r) \leq (km)^{1/2}/2$. If $W_P < (km)^{1/2}$, then the player does not move in HOLDBACK, and hence does not increase Φ. By Lemma 9, $\Delta\Phi \leq 2(km)^{1/2} W_A$, and the fact that $W_A \geq 1$ implies the inequality above. So we now assume that $W_P \geq (km)^{1/2}$. Then by Lemma 8, in step (1) Φ decreases by at least the distance that the player server moves before sending a temporary server, that is, at least $W_P - (km)^{1/2}$. By Lemma 9, the increase in Φ during step (2) is at most $2(km)^{1/2} W_A$. Thus $W_P + \Delta\Phi \leq (km)^{1/2} + 2(km)^{1/2} W_A \leq 3(km)^{1/2} W_A$. ■

HOLDBACK and other checkpoint algorithms for the circle can be extended to the line by thinking of the line as a circle with $k+1$ servers, one of which is fixed at 0. (This restriction applies to both the player and the adversary.) Whenever HOLDBACK tries to move the server at 0, instead the closest server behind 0 should be moved to the same spot. The analysis can be carried out in almost the same manner as above to prove that this version of HOLDBACK is $O((km)^{1/2})$-competitive for the checkpoint problem on the line.

8. Lower Bounds for Unequal Numbers of Servers

So far we have only compared online algorithms against offline algorithms with the same number of servers. Time was the resource used to measure competitiveness. It is natural to explore the space resource as well by allowing the online algorithms more servers than the offline algorithms. (Here we are following the lead of Manasse et al [11].) The next theorem also shows that our lower bounds are robust: strong (i.e., m^{ϵ}) lower bounds still hold even when the player is allowed k servers and the adversary only 1. Below we implicitly assume that k is much smaller than m, say k is $o(m^{\epsilon})$ for any fixed $\epsilon > 0$.

Theorem 6. *On-line algorithms with k servers can be no better than $\Omega(m^{\frac{1}{2^{k+1}-1}})$ competitive when compared to the 1-server optimal algorithm.*

Proof: The proof depends on a hierarchy of epoch sizes $\lceil m^{\alpha_1} \rceil, \lceil m^{\alpha_2} \rceil, \ldots, \lceil m^{\alpha_k} \rceil$, (given largest to smallest) defined by the recurrence:

$$\alpha_i = 2\alpha_{i+1} + \alpha_k, \qquad \alpha_k = \frac{1}{2^{k+1}-1}$$

or the closed form:

$$\alpha_i = \frac{2^{k+1-i}-1}{2^{k+1}-1}.$$

At the beginning of an epoch, the adversary is at an arbitrary location z_1. (Thus the adversary's strategy can be repeated for any number of epochs.) The adversary considers all possible ways of extending the current request sequence R_t by $\lceil m^{\alpha_1} \rceil$ requests in the arc $[z_1 + m^{\alpha_1}, z_1 + km^{\alpha_1}]$. If one of these (infinite number of) extensions causes the player to vacate the arc $(z_1 + km^{\alpha_1}, z_1]$, then the adversary chooses this extension, followed by a request at z_1, and processes the sequence by leaving its server at z_1. This results in player cost $\Omega(m)$ while the adversary cost is only $O(m^{2\alpha_1})$, giving the ratio claimed in the theorem.

On the other hand, if the player would maintain a server in $(z_1 + km^{\alpha_1}, z_1]$ for all possible extensions, then the adversary divides the epoch into $\Omega(m^{\alpha_1 - \alpha_2})$ subepochs of duration $\lceil m^{\alpha_2} \rceil$ each. The adversary starts the jth subepoch with its server at location $z_1 + m^{\alpha_1} + (j-1)(k-1)m^{\alpha_2}$. In each subepoch, the adversary considers all extensions of R_t by $\lceil m^{\alpha_2} \rceil$ requests in an arc of length $(k-2)m^{\alpha_2}$ starting m^{α_2} ahead of its server. If the player fails to maintain a server behind the adversary's location, a final request at that location forces the player's cost to $\Omega(m^{\alpha_1})$ while the adversary's cost is $O(m^{\alpha_2})$, thus giving the ratio claimed in the theorem. If the player would maintain a server, then that subepoch is further divided into subsubepochs of duration $\lceil m^{\alpha_3} \rceil$ comprising requests in subarcs of the subepoch's arc. In the ith level of this recursion a subarc has the form $[z_i + m^{\alpha_i}, z_i + (k-i+1)m^{\alpha_i}]$, where the choices of z_i are such that these subarcs are evenly spaced within a subarc of level $i-1$. The recursion terminates with a case identical to the single server proof of Section 3, since the player has $k-1$ confined servers and hence only one server free to process requests in the most deeply nested, zero-length subarc. ■

9. The Offline Problem

In this section we give algorithms for the problem of computing offline an optimal sequence of responses to a sequence of requests $r_1, r_2, \ldots, r_n$. For consistency our algorithms will be for the directed circle, though they can be easily modified for the directed line segment. Our algorithms are relatively slow, with a running time with an exponent of k. Although standard k-server problems have polynomial-time algorithms due to a reduction to minimum-cost flow [5], the checkpointing problem is quite nonlinear due to forking and excursions. We first consider the case $k = 1$.

Let $C_i(s)$ be the minimum cost of serving requests $r_1, r_2, \ldots, r_i$ and leaving the server at position $s \in [0, m)$. We have the recurrence

$$C_i(s) = \min\{C_{i-1}(s) + d(s, r_i), C_{i-1}(r_i) + d(r_i, s)\},$$

where the first term corresponds to the option of leaving the server at s and serving the ith request with the temporary server and the second term corresponds to the option of leaving the server at r_i after the previous request and then moving to position s after serving the ith request. Other possibilities, such as leaving a server at a position s' and then moving from s' to r_i to s, are dominated by these two options. A reasonable initial condition is $C_0(s) = s$.

Lemma 10. *$C_i(s)$ is a continuous, piecewise-linear function with $i+1$ pieces and maximum slope 1.*

Proof: This statement is trivially true for $C_0(s)$, so assume that it holds inductively for $C_{i-1}(s)$. Then $C_{i-1}(s) + d(s, r_i)$ has maximum slope 0, i pieces, and a discontinuity only at $s = r_i$. $C_{i-1}(r_i) + d(r_i, s)$ has slope identically 1 and is continuous with $C_{i-1}(s) + d(s, r_i)$ at $s = r_i$. That is, $C_{i-1}(s) + d(s, r_i)$ to the left of r_i and $C_{i-1}(r_i) + d(r_i, s)$ to the right of r_i form a continuous function. Thus for exactly some nonempty interval to the right of r_i, the minimum of $C_{i-1}(s) + d(s, r_i)$ and $C_{i-1}(r_i) + d(r_i, s)$ will be achieved by $C_{i-1}(r_i) + d(r_i, s)$. So the lemma holds for $C_i(s)$. ■

Let p denote the number of distinct locations among $r_1, r_2, \ldots, r_n$. Obviously $p \leq n$; below we state our running times in terms of p rather than n as in many applications $p \ll n$.

We now show how to store the functions $C_i(s)$ implicitly in a complete binary tree T with $p+1$ leaves in which each node stores a linear function of s.

Lemma 11. *The functions $C_i(s)$ can be stored in a binary tree data structure such that the entire sequence of updates from $C_0(s)$ to $C_n(s)$ takes time $O(n \log p)$.*

Proof: Each leaf of tree T corresponds to a minimal half-open interval $[r_i, r_j)$ between request locations on $[0, m)$. These intervals are sorted in left-to-right order across the tree. Each internal node v in T corresponds to a half-open interval, *segment*(v), that is the union of all segments corresponding to leaves in v's subtree. As in Bentley's segment tree [12], an interval of $[0, m)$ with endpoints at request locations corresponds to $O(\log m)$ *basic nodes*, those nodes v such that *segment*(v) is contained in the interval but *segment*$($*parent*$(v))$ is not.

As usual, each node v of T stores its left and right endpoints, $v.L$ and $v.R$, and its midpoint $v.M$, that is, the right endpoint of its left child. Each node also stores a linear function $v.a \cdot s + v.b$. $C_i(s)$ is computed by summing the values of these functions on the path from leaf s to the root. In addition, v stores $v.LC$, $v.MC$, and $v.RC$, the partial sums up to v from, respectively, the leftmost leaf, leftmost leaf in the right child subtree, and rightmost leaf. Finally, v stores a boolean $v.O$ that is true exactly when all linear functions in v's subtree are identically 0. Initially all linear functions are identically 0, except at the root of T which stores the function s.

Updating $C_{i-1}(s)$ to give $C_i(s)$ involves the following steps.

1. Compute $C_{i-1}(r_i)$ by summing the values $v.a \cdot r_i + v.b$ for all v along the path from leaf r_i to the root. Call this value C.

2. Add $-s + r_i$ to the linear function at each basic node of $[0, r_i)$, and update $v.LC$, $v.MC$, and $v.RC$ for each v along the paths from these basic nodes to the root.

3. Add $-s+m+r_i$ to the linear function at each basic node of $[r_i, m)$, and update $v.LC$, $v.MC$, and $v.RC$ fields as above.

4. Compute the leaf interval containing the point $z \in [0, m)$ for which $C + d(r_i, z) = C_{i-1}(s) + d(s, r_i)$. Let z' be the right endpoint of this interval.

5. If $z' \in [r_i, m)$, then set all linear functions throughout the subtrees of the basic nodes of $[r_i, z')$ to zero. Then set the linear functions at each basic node v of $[r_i, z')$ to be such that the sum from v to the root is the function $s - r_i + C$. If $z' \in [0, r_i)$, then follow the procedure above for the interval $[r_i, m)$. In addition, zero all functions in $[0, z')$ and set the functions at each basic node v of $[0, z')$ to be such that the sum from the root to v is $s + m - r_i + C$.

Steps 4 and 5 above require further explanation. Step 4 can be accomplished by a binary search for z. Let $C_v(s)$ be the linear function that is the sum of the linear functions from the root of T to v. Assume v is a node such that r_i is not interior to *segment*(v). (One can find all maximal nodes satisfying this condition in time $O(\log p)$.) Then z lies in *segment*(v) if and only if $C_v(v.L) + v.LC \le C + d(r_i, v.L)$ and $C_v(v.R) + v.RC > C + d(r_i, v.R)$. A similar $O(1)$-time test determines whether z lies to the left or right of midpoint $v.M$.

We describe step 5 for the case $z' \in [r_i, m)$. The other case is similar. First we search for all nodes v with *segment*(v) in $[r_i, z')$ such that $v.O$ is false. At these nodes, $v.a$ and $v.b$ are set to zero, and $v.O$ is set true. The settings of $v.O$ along paths to the root must also be updated. Then in a second phase, the linear functions at basic nodes v of $[r_i, z')$ are set to $-C_v(s) + s - r_i + C$. Fields $v.O$ must again be updated up to the root.

The time analysis for steps 1 to 4 follows from the usual logarithmic time bounds for segment trees. The time spent in step 5 may be much larger than $O(\log p)$ for any given step, but a bound of $O(n \log p)$ for all n steps 5 follows from the observation that the time to zero fields in the segment tree is of the same order as the time to set those same fields. ■

Theorem 7. *For $k = 1$, the offline problem can be solved in time $O(n \log p)$, where $p \leq n$ is the number of distinct request locations.*

Proof: We build the binary tree described above in time $O(p \log p)$. Then we process the sequence of requests $r_1, r_2, \ldots, r_n$ by updating the tree as in Lemma 11. While doing this, we build a list $z_1, z_2, \ldots, z_n$, giving the z (or z') values for each update. The optimal sequence of server positions $s_1, s_2, \ldots, s_n$ occupied after requests $r_1, r_2, \ldots, r_n$ is then computed in reverse order. The minimum value of $C_n(s)$ is necessarily achieved at $s_n = r_n$; this is the optimal cost of handling requests $r_1, r_2, \ldots, r_n$. If s_n lies in the wrapped-around interval $[r_{n-1}, z_{n-1})$, then the last motion of the server is a move from r_{n-1} to r_n, and $s_{n-1} = r_{n-1}$. If s_n lies in $[z_{n-1}, r_{n-1})$, then r_{n-1} should be served with an excursion from s_n, and $s_{n-1} = s_n$. Similarly, if $s_{n-1} \in [z_{n-2}, r_{n-2})$, then r_{n-1} is also served with an excursion. In this way we recover the optimal sequence of server moves. ■

We now give a theorem for the case of more than one server. We present a straightforward algorithm, though we expect that a single factor of $O(p)$ in the running time can be replaced by $O(\log p)$ using a suitable multi-dimensional data structure. The following technical lemma limits the number of cases we must consider in the general dynamic programming algorithm.

Lemma 12. *An optimal sequence of moves for requests $r_1, \ldots, r_n$ can be rearranged into an optimal sequence that uses only moves of the following form: server i closest to the request is moved 0 or more distance towards the request, the temporary server serves the request from distance 0 or greater, and 0 or more other servers are forked to occupy locations traversed by i or the temporary server.*

Proof: Assume we have an optimal sequence of moves in which server i and server j both make smooth (that is, nonforking) motion in response to a request r_l. Assume that server i or the temporary leaving from a position occupied by server i is the server that actually serves r_l. Then, without increasing the cost of the sequence of moves, we may delay the motion of j until the first request served by one of the following: j itself, a server forked to a position occupied by j, or the temporary server launched from a position occupied by j. ■

Theorem 8. *For fixed $k \geq 2$, the offline problem can be solved in time $O(np^k)$.*

Proof: Let $C_i(s_1, s_2, \ldots, s_k)$ be the minimum cost of serving requests $r_1, r_2, \ldots, r_i$ and leaving servers at locations $s_1 \leq s_2 \leq \ldots \leq s_k$. Let $C_i(s_1, s_2, \ldots, s_l, *)$, with $l \leq k$, denote the minimum cost of serving requests $r_1, r_2, \ldots, r_i$ and leaving servers at locations $s_1 \leq s_2 \leq \ldots \leq s_l$ and any other $k - l$ locations. Note $C_i(s_1, \ldots, s_k, *) = C_i(s_1, \ldots, s_k)$. We have the recurrence

$$\begin{aligned} C_i(s_1, \ldots, s_k) &= \min\Big\{ C_{i-1}(s_1, \ldots, s_{j-1}, r_i, s_{j+1}, \ldots, s_k) + d(r_i, s_j), \\ &\qquad \min_{h<j}\{C_{i-1}(s_1, \ldots, s_h, s_j, \ldots, s_k, *) + d(s_h, r_i)\} \Big\}, \end{aligned}$$

where j is such that $s_{j-1} \leq r_i \leq s_j$. The first part of the minimization above corresponds to the option of leaving a server at r_i and then moving it without forks to s_j. The second part corresponds to serving request r_i with a temporary server from location s_h while forking processors to occupy positions $s_{h+1}, \ldots, s_{j-1}$. By Lemma 12, other possibilities are dominated by these two options. Let S denote an ordered sequence of k request locations in $[0, m)$. We also have the recurrence

$$C_i(s_1, \ldots, s_l, *) = \min\{C_i(S) \mid (s_1, \ldots, s_l) \text{ is a subsequence of } S\},$$

and the initial condition

$$C_i(s_1, \ldots, s_l, *) = s_l.$$

We store the values of $C_i(s_1, \ldots, s_l, *)$ in a sequence of k arrays, $A_1, A_2, \ldots, A_k$. Array A_j is a j-dimensional array storing all values of $C_i(s_1, \ldots, s_j, *)$. Since k is assumed fixed, indexing into this sequence of arrays and updating an entry takes time $O(1)$.

Array A_k is updated using the first recurrence given above. When $C_i(s_1, \ldots, s_k)$ is changed, 2^k (which is $O(1)$) entries in arrays $A_1, \ldots, A_{k-1}$ must also be updated. This procedure implicitly implements the second recurrence. Altogether we obtain time $O(np^k)$ for computing $C_n(s_1, \ldots, s_k)$. As in the algorithm for $k = 1$, the optimal sequence of moves can be reconstructed by maintaining a record of all minimizing choices. ■

10. Conclusions

We have explored adaptive online schemes for locating checkpoints. To do so we introduced a server problem that includes several nonstandard features: a fixed cost per request, one-way motion, excursions, and forking. Including the fixed cost enabled us to differentiate algorithms that would otherwise have simply been declared noncompetitive. One-way motion and excursions taken together raise the optimal competitiveness from k (the number of servers [5, 11]) to about $m^{1/3}$, where m is in effect the size of the playing field. Forking further raises this bound to about $m^{1/2}$. A number of interesting open problems remain:

- Does there exist an $O(m^{1/3})$-competitive algorithm for locating checkpoints in the case $k = 2$? (This question raises the important issue of whether forking is of any use to the player. For $k = 2$, we can prove a lower bound of $\Omega(m^{1/2})$ on the competitiveness of any online algorithm that does not fork.)

- Except for $k = 2$, our upper and lower bounds match in their dependence on the dominant factor m. There remain, however, constant gaps and gaps depending on k. Can these be closed?

- What happens to our bounds if we disallow excursions? That is, memory is now assumed homogeneous.
- What is the effect of allowing forking on other server problems?

For some problems, most notably accessing a linear list, competitive analysis seems to give "the right answer"—that is, it leads to an algorithm that arguably dominates all others. For the checkpointing problem, the situation is less clear. We believe that competitive analysis has demonstrated the utility of an initially rapid, then increasingly slow, approach to a repeated request location. We also think that it has invalidated some initially attractive algorithms, such as one that always moves halfway towards a request. On the other hand, due to its emphasis on worst-case sequences, competitive analysis may have led us to overly conservative algorithms. In practice one would probably want to move a checkpoint closer than the distance $(km)^{1/2}$ prescribed by HOLDBACK.

In fact the choice of a practical algorithm, whether HOLDBACK, TWO-PHASE (which is slightly more aggressive), or something else, should depend on how "adversarial" are the expected request sequences. In applications such as debugging or physical simulation, there may be a small number of "hot spots" and users may often step backwards in time. In such situations request sequences may indeed appear quite adversarial.

It would be interesting to investigate the checkpoint location problem using other styles of analysis, such as probabilistic analysis assuming random (possibly correlated) requests.

Acknowledgements

We would like to thank Mike Paterson and Howard Karloff for some valuable discussions.

References

[1] S. Ben-David, A. Borodin, R. Karp, G. Tardos, and A. Wigderson, On the Power of Randomization in Online Algorithms, these proceedings.

[2] P. Berman, H. Karloff, and G. Tardos, A Competitive Three-Server Algorithm, *1st ACM-SIAM Symp. on Discrete Algorithms*, 1989.

[3] A. Borodin, N. Linial, and M. Saks, An Optimal Online Algorithm for Metrical Task Systems, *19th ACM Symp. on Theory of Computing*, 1987.

[4] A. Calderbank, E. Coffman, and L. Flatto, Sequencing Problems in Two-server Systems, *Math. Oper. Research* 10, 1985, 585-598.

[5] M. Chrobak, H. Karloff, T. Payne, and S. Vishwanathan, New Results on Server Problems, *1st ACM-SIAM Symp. on Discrete Algorithms*, 1989.

[6] M. Chrobak and L. Larmore, An Optimal On-Line Algorithm for k Servers on Trees, manuscript, UC - Riverside, 1989.

[7] D. Coppersmith, P. Doyle, P. Raghavan, and M. Snir, Random Walks on Weighted Graphs, and Applications to On-line Algorithms, these proceedings.

[8] E. Fiala and D. Greene. Data Compression with Finite Windows, *CACM 32*, April 1989. 490-505.

[9] A. Karlin. M. Manasse. L. Rudolph. and D. Sleator. Competitive Snoopy Caching. *Algorithmica* 3. 1988, 79-119.

[10] R. Korf, Complexity of Reverse Execution, manuscript, 1981.

[11] M. Manasse. L. McGeoch, and D. Sleator, Competitive Algorithms for On-line Problems, *20th ACM Symp. on Theory of Computing*, 1988.

[12] F. P. Preparata and M. I. Shamos, *Computational Geometry: An Introduction*, Springer-Verlag. 1985.

[13] P. Raghavan and M. Snir, Memory vs. Randomization in Online Algorithms, *ICALP*, 1989.

[14] D. Sleator and R. Tarjan, Amortized Efficiency of List Update and Paging Rules, *CACM* 28, February 1985. 202-208.

[15] J. Storer. *Data Compression*, Computer Science Press, 1988.