Report Number: WUCS-96-20

1996-01-01

Optimal Solution of Off-line and On-line Generalized Caching

Saied Hosseini-Khayat and Jerome R. Cox

Network traffic can be reduced significantly if caching is utilized effectively. As an effort in this direction we study the replacement problem that arises in caching of multimedia objects. The size of objects and the cost of cache misses are assumed non-uniform. The non-uniformity of size is inherent in multimedia objects, and the non-uniformity of cost is due to the non-uniformity of size and the fact that the objects are scattered throughout the network. Although a special case of this problem, i.e. the case of uniform size and cost, has been extensively studied, the general case needs a great... Read complete abstract on page 2.

Optimal Solution of Off-line and On-line Generalized Caching

Saied Hosseini-Khayat and Jerome R. Cox

Complete Abstract:

Network traffic can be reduced significantly if caching is utilized effectively. As an effort in this direction we study the replacement problem that arises in caching of multimedia objects. The size of objects and the cost of cache misses are assumed non-uniform. The non-uniformity of size is inherent in multimedia objects, and the non-uniformity of cost is due to the non-uniformity of size and the fact that the objects are scattered throughout the network. Although a special case of this problem, i.e. the case of uniform size and cost, has been extensively studied, the general case needs a great deal of study. We present a dynamic programming method of optimally solving the off-line and on-line versions of this problem, and discuss the complexity of this method.

Optimal Solution of Off-line and On-line Generalized Caching

Saied Hosseini-Khayat and Jerome R. Cox

WUCS-96-20

July 1996

Applied Research Laboratory
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Optimal Solution of Off-line and On-line Generalized Caching

Saied Hosseini-Khayat * and Jerome R. Cox, Jr.

Washington University in St. Louis

Abstract. *Network traffic can be reduced significantly if caching is utilized effectively. As an effort in this direction we study the replacement problem that arises in caching of multimedia objects. The size of objects and the cost of cache misses are assumed non-uniform. The non-uniformity of size is inherent in multimedia objects, and the non-uniformity of cost is due to the non-uniformity of size and the fact that the objects are scattered throughout the network. Although a special case of this problem, i.e. the case of uniform size and cost, has been extensively studied, the general case needs a great deal of study. We present a dynamic programming method of optimally solving the off-line and on-line versions of this problem, and discuss the complexity of this method.*

Key words: Generalized caching, network traffic, network caching, file caching, optimal replacement, replacement algorithm.

I. Introduction

Caching is an effective performance enhancement technique that has been used in computer systems for decades. A cache is a temporary store for frequently accessed data and its purpose is twofold: to provide fast access to data, and to reduce traffic between the main store and the consumer of data. The ongoing information revolution is creating new applications which can benefit from caching. Networking in conjunction with caching will allow information-on-demand providers to reduce storage and transmission costs. For instance, on-line digital libraries organized as a network of archive nodes and cache nodes interconnected with high speed links will serve a large number of users. The cache nodes will decrease the network load by taking advantage of temporal and spatial locality of requests by the users. Video caches in video-on-demand systems, for example as proposed in [7, 8], will increase the utilization of the network by caching popular programs ahead of time during off-peak hours and reduce network traffic by serving repeated requests by different subscribers. Site-level as well as user-level document caching tremendously benefits the World Wide Web services [3, 5], and research in this area is ongoing. Distributed image databases can also utilize caching to reduce server and network load. A common feature of these new applications is the non-uniform size of cached objects as well as the non-uniform cost of cache misses. The cost, whether it is the waiting time perceived by users or the traffic generated in the network, normally depends on the size of items, the distance they travel and other communication link parameters. Also given is the fact that cache space is always limited. Therefore deciding which item(s) to replace when a new one arrives is an important and interesting optimization problem. This problem has been adequately studied in memory paging [10, 11], however with the non-uniformity assumption that holds in the network environment it deserves a great deal of new research. In this paper, we present a method of optimally solving the replacement problem. Section II and III present the notations and definition of the problem. In Section IV we discuss the off-line case, and solve the problem by transforming it into the shortest path problem. Section V presents the optimal solution of the on-line case. Finally Section VI contains our conclusions.

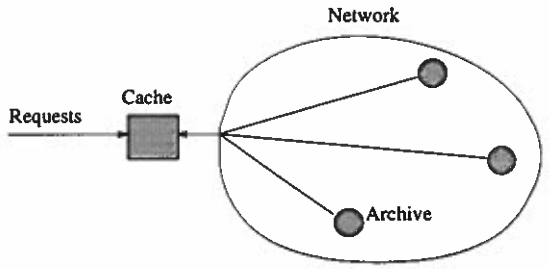

Figure 1: A network cache

*Applied Research Laboratory, Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, St. Louis MO 63130-4899. email: saied@arl.wustl.edu. Tel:(314) 935-4460.

II. Notations and Assumptions

Given is a finite universe of objects $\mathbb{U}$ represented by $\{1, 2, \ldots, N\}$. For each object i there is a size $a_i \in \mathbb{R}^+$ and a cost $c_i \in \mathbb{R}^+$. A *cache* is a set $\mathcal{B} \subset \mathbb{U}$ such that

$$\sum_{i \in \mathcal{B}} a_i \leq B \ ,$$

where B is the capacity of the cache.

A *sequence of requests* $\rho : \mathbb{N} \to \mathbb{U}$ is denoted by $\sigma_1, \sigma_2, \ldots, \sigma_m$. By $\sigma_k = i$ we mean that item i is requested at time k. The *maximum cost* of ρ is

$$W_{max}(\rho) \triangleq \sum_{k=1}^{m} c_{\sigma_k} \ .$$

The *state* of a cache is the set of objects it contains and may change in response to requests. Let $\mathcal{B}_k$ denote the state of cache $\mathcal{B}$ at time k. The *state space* of a cache is

$$\mathbb{B} = \{\mathcal{S} \subset \mathbb{U} \mid \sum_{i \in \mathcal{S}} a_i \leq B\}.$$

Note that $\emptyset \in \mathbb{B}$. The set $\mathcal{S}_j$ denotes the collection of all states containing element j. The *state sequence* $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_m$ denotes consecutive states of the cache in response to a sequence of m requests, where $\mathcal{B}_0$ is the initial state and $\mathcal{B}_m$ is the final state.

A *caching policy* A takes a request sequence ρ and a cache of size B at initial state $\mathcal{B}_0$, produces a state sequence $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m$ and incurs a cost $W(A, \rho, B)$ which is defined later. We use $W(\{\mathcal{B}_k\}, \rho, B)$ to denote the cost of a particular state sequence. The *miss index* (normalized cost) is

$$M(A, \rho, B) \triangleq \frac{W(A, \rho, B)}{W_{max}(\rho)} \ .$$

Note that: $0 \leq M \leq 1$.

In this paper, a single cache and non-modifiable, non-dividable cache objects are considered. We assume that requests to the cache must be served in the order of arrival and every missing item is loaded into cache at the time of request. A miss has a penalty equal to the cost of the missing item, and a hit costs zero. Every time a new item is loaded, one or more items may have to be purged. No cost is incurred for purging an item.

III. Definition: The Generalized Caching Problem

Given a universe $\mathbb{U}$ of items with sizes $a_1, a_2, \ldots, a_n$ and costs $c_1, c_2, \ldots, c_n$, a cache of size B in initial state $\mathcal{B}_0$, and a request sequence $\rho = \sigma_1 \sigma_2 \ldots \sigma_m$, find a state sequence $\{\mathcal{B}_k\}_{k=1}^{m}$ such that

i. for all $i = 1, 2, \ldots, m$, we have

$$\sum_{j \in \mathcal{B}_i} a_j \leq B,$$

ii. for all $i = 1, 2, \ldots, m$, we have

$$\mathcal{B}_i \ = \ \begin{cases} (\mathcal{B}_{i-1} - \mathcal{E}_i) \cup \{\sigma_i\} & \text{if } \sigma_i \notin \mathcal{B}_{i-1} \\ \mathcal{B}_{i-1} & \text{if } \sigma_i \in \mathcal{B}_{i-1} \end{cases},$$

where $\mathcal{E}_i \subset \mathcal{B}_{i-1}$.

iii. and the cost $W(\{\mathcal{B}_k\}, \rho, B) = \sum_{k=1}^{m} \delta_k \ c_{\sigma_k}$ is minimized, where

$$\delta_i \ \triangleq \ \begin{cases} 0 & \text{if } \sigma_i \in \mathcal{B}_{i-1} \\ 1 & \text{if } \sigma_i \notin \mathcal{B}_{i-1} \end{cases}.$$

IV. Off-Line Solution

This section present the solution of off-line optimal replacement. In the off-line case, it is assumed that the sequence of requests is known in advance. This may be true, for example, in applications for which an advance schedule of requests is available. The solution to a special case of this problem with items of uniform size and cost was shown [2, 6] to be a policy that replaces, among all items currently in cache, an item whose next request comes last in the future (Longest Forward Distance Policy or LFD). First we show that this policy is not optimal in the general case. Then we use the Principle of Optimality and dynamic programming [4] to solve the general problem.

A. LFD is not Optimal

LFD is an off-line policy that replaces an item in cache whose next request lies furthest in future. The following example shows that this policy is not optimal in general. Consider $\mathbb{U} = \{1, 2, 3, 4\}$, $a_1 = a_2 = a_3 = a_4 = 1$, $c_1 = c_2 = 1$, $c_3 = 5$, $c_4 = 10$. Let $B = 2$ and $\rho = 1, 4, 3, 2, 1, 4$. Table A shows two different policies: LFD and another policy A which

σ_k	$\mathcal{B}_k(lfd)$	$\delta_k(lfd)c_{\sigma_k}$	$\mathcal{B}_k(A)$	$\delta_k(A)c_{\sigma_k}$
1	1	1	1	1
4	1,4	10	1,4	10
3	1,3	5	3,4	5
2	1,2	1	2,4	1
1	1,2	0	1,4	1
4	1,4	10	1,4	0

Table 1: LFD costs more than Policy A

incurs less cost than LFD: $W(A,\rho,B) = 18 < 27 = W(LFD,\rho,B)$. Therefore LFD does not produce optimal results in general.

This example demonstrated the case of uniform size and non-uniform cost. It is easy to show that LFD also fails when only the sizes or both sizes and costs are non-uniform.

B. Principle of Optimality

We show that the Principle of Optimality holds in our problem. This allows us to use dynamic programming in finding the optimal state sequence.

THEOREM 1. **Principle of Optimality.** *Given a universe $\mathbb{U}$ of items, a cache $\mathcal{B}$ in initial state $\mathcal{B}_0$ and a sequence $\{\sigma_k\}_{k=1}^m$, let $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_m$ be an optimal state sequence. Suppose $\mathcal{B}_i = \mathcal{S}$ for some $i \in \{1, 2, \ldots, m\}$. Then among all state sequences starting at $\mathcal{B}_0$, fulfilling $\{\sigma_k\}_{k=1}^i$ and ending $\mathcal{S}$ at time i, the sequence $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{i-1}, \mathcal{S}$ has the least cost. Also among all state sequences starting at $\mathcal{S}$ and fulfilling $\{\sigma_k\}_{k=i+1}^m$, the sequence $\mathcal{S}, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_m$ has the least cost.*

Proof. Suppose $\mathcal{B}_0, \mathcal{B}'_1, \ldots, \mathcal{B}'_{i-1}, \mathcal{S}$ fulfills $\{\sigma_k\}_{k=1}^i$ and has a lower cost than $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{i-1}, \mathcal{S}$. Then $\mathcal{B}_0, \mathcal{B}'_1, \ldots, \mathcal{B}'_{i-1}, \mathcal{S}, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_m$ has an overall cost lower than $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{i-1}, \mathcal{S}, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_m$. This is a contradiction. Therefore the first claim holds. Now let $\mathcal{S}, \mathcal{B}'_{i+1}, \ldots, \mathcal{B}'_m$ fulfill $\{\sigma_k\}_{k=i+1}^m$ and have a cost lower cost than $\mathcal{S}, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_m$, then $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{i-1}, \mathcal{S}, \mathcal{B}'_{i+1}, \ldots, \mathcal{B}'_m$ has an overall cost lower than $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_{i-1}, \mathcal{S}, \mathcal{B}_{i+1}, \ldots, \mathcal{B}_m$, which is another contradiction. This proves the second claim. □

C. Dynamic Programming Method

In this section we transform the caching problem into the problem of finding a shortest path in a special class of graphs. Then a customized dynamic programming algorithm is presented that solves this problem.

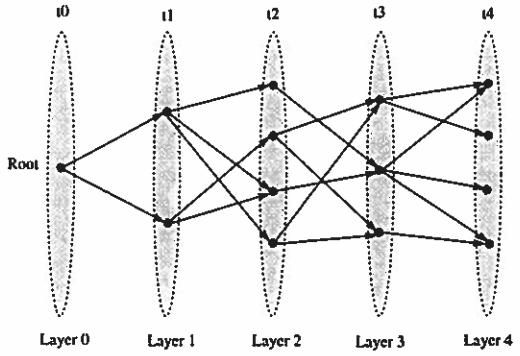

Figure 2: Layered DAG of a cache

First note the following observations: Caching is a multi-stage decision process and can be represented as a multi-layer directed acyclic graph (DAG) (Figure 2) whose nodes are cache states at particular times, and whose arcs are transitions between states. The initial state is the *root* of this graph. Every node may have multiple parents and multiple children (Figure 3). The associated DAG has $m+1$

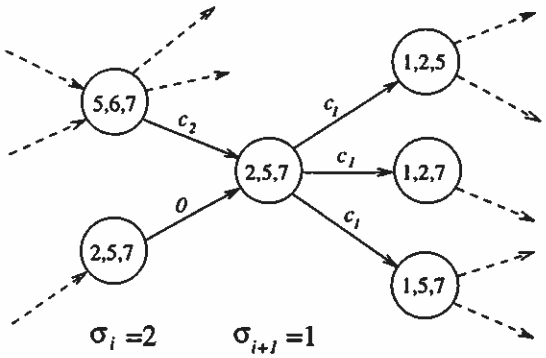

Figure 3: State transitions

layers (including the root layer containing the root) corresponding to m requests with nodes in layer i being all states reachable at time i from the root. The root nodes in layer $i \geq 1$ are children of nodes in the previous layer. A cost is associated with each arc. The cost is zero if the transition is due to a hit, and is equal to the cost of the current request if the transition is due to a miss (Figure 3). There are possibly multiple paths from the initial state to each node. Every path from the root to a node in the last layer (a terminal node) corresponds to a complete state sequence and the accumulated cost of arcs along each path is the cost of corresponding state sequence. An optimal path is one such path that has the least cost. Thus we are looking for the

"shortest" path from the root to any of the terminal nodes. Let u be a node in layer $i \geq 1$ of the DAG. If u is on an optimal path, then by the Principle of Optimality the optimal path must coincide with a path from the root to u that has the least cost. Therefore all but one path from the root to u are redundant and can be pruned.

Let us introduce some notations: Denote the set of nodes in layer i by V_i, and the set of parents of a node v by $\Lambda(v)$. Assign to every node v a cost $w(v)$, and a *special parent* $\lambda(v)$ (excluding the root which has no parents). Denote the cost of an arc from a node v to a node v' by $w(v, v')$. Lead by the above observations, let $w(v)$ be the least accumulated cost from the root to node v. This can be determined recursively as follows:

$$w(v) = \min_{v' \in \Lambda(v)} [w(v') + w(v', v)] .$$

Also let the special parent $\lambda(v)$ of node v be a parent that minimizes $[w(v') + w(v', v)]$. This fact is used in the following algorithm which proceeds layer by layer, generates nodes in each layer, and determines the costs and special parents of nodes. Finally it picks in the final layer a node with the least cost. This is an optimal final state and all its special ancestors back to the root form an optimal state sequence.

Algorithm:

1. Start from the initial cache state *root*. ($root \in V_0$).

2. For each node in the current layer, generate its children that correspond to the current request. Skip this step if this is the final layer V_m.

3. For each node v in the current layer find its cost

$$w(v) = \min_{v' \in \Lambda(v)} [w(v') + w(v', v)] ,$$

   and its special parent $\lambda(v)$ that minimizes $[w(v') + w(v', v)]$. ($w(root) = 0$, $\lambda(root) = null$.)

4. If the current layer is not final, proceed to the next layer and repeat steps 2, 3, 4.

5. Find a node v^* in the final layer with the least cost.

6. Return $\{v^*, \lambda(v^*), \lambda(\lambda(v^*)), \ldots, root\}$ and $w(v^*)$. End.

The returned value $w(v^*)$ is the optimal caching cost, and the sequence of vertices $\{v^*, \lambda(v^*), \lambda(\lambda(v^*)), \ldots, root\}$ is an optimal state sequence in the opposite order. This algorithm can be implemented in such a way that for each node only its optimal ancestors are kept and the rest deleted thus saving memory space. The above algorithm has some resemblance to the relaxation method of finding a shortest path in a graph [4, page 520] and the Viterbi convolutional decoding algorithm [9, page 331].

D. Discussion

It was shown how to obtain an optimal solution to the generalized off-line caching. It is worthwhile to discuss the efficiency of this algorithm. Notice that the algorithm basically explores a DAG which is associated with the caching problem at hand. Therefore the computation time is roughly proportional the size of the DAG. Also note that the breadth of the DAG grows at the beginning and then eventually saturates and will never exceed $\max_{j \in \mathbb{L}} |\mathcal{S}_j|$ the maximum number of nodes in a layer. Therefore the maximum breadth of the DAG is independent of m the length of request sequence, implying that that computation time per layer is asymptotically constant with m. The depth (i.e. the number of layers) of the DAG, on the other hand, is linearly dependent on only m. Thus the computation time of the algorithm, if everything else is fixed, is $O(m)$. However if we increase the size of the universe $\mathbb{U}$ or the size of cache B, the breadth grows too rapidly. This is a result of explosive growth of the state space, which is characteristic of combinatorial problems. Therefore, unless a more efficient solution is found, one must resort to heuristic methods to solve large scale generalized caching problems.

We implemented this algorithm and performed two types of experiments. First, we were interested in comparing the performance of LFD (non-optimal) and the dynamic programming method (optimal). Our experiments showed that although LFD is not optimal, it performs relatively close to optimal in typical cases. For example, we picked a universe of 20 items with sizes 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 and costs 2, 2, 5, 5, 5, 10, 10, 10, 10, 10, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, respectively. The cache size was 20. Both LFD and dynamic programming method were applied to 100 randomly generated sequences of length 1000, and their miss indices were

compared. The difference was less than 7% of the optimal value.

In an effort to approximate the optimal, another set of experiments was carried out to observe the effect of deleting all except N best (least cost) nodes in every layer as the algorithm proceeds. The result was surprisingly good. As an example, in one experiment we used the same universe of objects as above, generated a random request sequence of length 1000 (with the maximum cost $W_{max}(\rho) = 5593$) and set the cache size to 30. First we ran the optimal method and computed the optimal cost. The maximum breadth of the DAG of this problem was 7523 nodes. Then a modified algorithm was run which performed node deletion as described above with different values of parameter N. The results is tabulated below.

N	Cost	Deviation
2	2306	75%
5	1883	42%
10	1711	30%
20	1554	18%
50	1404	6%
100	1350	2%
200	1327	0.6%
500	1322	0.2%
1000	1319	0%
∞	1319	optimal

Table 2: Percentage of deviation vs. N

This table shows that even when the breadth of the DAG is squeezed to 1.33% of its maximum value (by keeping $N = 100$ best nodes in each layer) the computed cost deviates from the optimal by only 2% ! This indicates that computation time can be reduced drastically if a slight increase in cost is acceptable.

V. On-Line Solution

Now we present the solution of on-line optimal replacement. In this case, the requests arrive one at a time and the algorithm must decide before the next request arrives. An algorithm is called *on-line* if replacement decisions are made without knowledge of future requests. In the worst case, an on-line algorithm can incur the maximum cost $W_{max}(\rho)$ of a sequence. However if requests are generated according to a probabilistic model, the problem will become that of finding a policy that minimizes the expected cost of caching a sequence. In this paper we assume the following:

Assumption: *Requests are generated according to an irreducible aperiodic Markov chain with transition probability matrix* $[p_{ij}]$.

This can be a good model for World Wide Web browsing in which the next request is highly dependent on the current request. Also it contains the independent request model as a special case. With the above assumption, caching can be viewed as a discrete-time Markov decision process [12] in the following way: Let the state space of our system be $\mathbb{S} = \mathbb{B} \times \mathbb{U}$. A system state at any time is an ordered pair of the current cache state and the current request. Corresponding to every state $s \in \mathbb{S}$ there is a set of actions $\mathcal{A}(s)$. An action $a \in \mathcal{A}(s)$ is described by a set of items to be purged from cache and its cost $c(a, s)$ is the total cost of items it purges. A policy is a set of actions, one for each system state. It prescribes one action when it sees the system in a particular state. If at a decision epoch the system is in state $s = (\mathcal{S}, i)$ and action $a \in \mathcal{A}(s)$ is chosen, then regardless of the past history of the system, the following happens:

(a) An immediate cost $c(s, a)$ is incurred.

(b) At the next decision epoch the system will be in state $(\mathcal{S}', j)$ with probability p_{ij}, and such that $\mathcal{S}' = (\mathcal{S} - \mathcal{A}(s)) \cup \{i\}$.

The value iteration algorithm [12, page206] computes recursively for $k = 1, 2, \ldots$ the value function

$$Y_k(s) = \min_{a \in \mathcal{A}(s)} \left\{ c(s, a) + \sum_{s' \in \mathbb{S}} p_{ij} Y_{k-1}(s') \right\} , s \in \mathbb{S} ,$$

starting with $Y_0(s) = 0,\ s \in \mathbb{S}$. The quantity $Y_k(s)$ is the minimal total expected cost of actions when the system starts at s and continue for k decisions. From the above we obtain an optimal policy, i.e. one action for each system state, that minimizes the expected total cost of actions over a finite horizon of k decision epochs. Let us define the minimum expected cost of actions per request as

$$g(s) \triangleq \lim_{k \to \infty} \frac{1}{k} Y_k(s),\ s \in \mathbb{S}$$

Notice that minimizing the value function (the total cost of purged items) is not the objective of caching. Rather the goal is to minimize the total cost of

loaded items. However, in the long run since the total cost of purged items is equal to the total cost of loaded items, then $g(s)$ is also the minimum expected cost per request of caching. In practice, one is usually satisfied with a policy whose expected cost per request is sufficiently close to $g(s)$. Define

$$M_k \triangleq \max_{s \in \mathbb{S}} [Y_k(s) - Y_{k-1}(s)] ,$$

$$m_k \triangleq \min_{s \in \mathbb{S}} [Y_k(s) - Y_{k-1}(s)] .$$

If k is increased until $0 \leq M_k - m_k \leq \epsilon m_k$, then by Theorem 3.4.1 in [12] the expected cost per request cannot deviate more then $100\epsilon\%$ from $g(s)$.

A. Discussion

It was shown that stochastic dynamic programming can solve the optimal on-line caching problem. The same approach was used in [1] for the memory paging problem. However, the drawback in both cases is the rapid growth of the state space. For this and other reasons, heuristic methods such as LRU (Least Recently Used) and LFU (Least Frequently Used) have been used in caching for decades. Therefore heuristic solutions must be used in large scale caching problems. We are studying generalized heuristics that take into account the size and cost of items.

VI. Conclusion

We showed that the off-line and on-line generalized caching can be solved optimally by means of dynamic programming. This approach, however, due to the explosive growth of its state space, has limited practicality. We think that it is unlikely, as in many other combinatorial problems, to find a practical optimal solution, and therefore efforts must be directed towards finding effective and practical heuristics. One such heuristic in the off-line case can be the LFD algorithm which is optimal only when the sizes and costs are uniform. Our experiments show that LFD performs close to optimal when the request sequences are not pathological. Another finding is that a trade-off between computation time and optimality is possible when, in the off-line method, all but N best paths are omitted as the algorithm proceeds. This may be a fruitful direction of research toward approximate solutions.

References

[1] V. A. Aho, P. J. Denning, J. D. Ullman, "Principles of Optimal Page Replacement," *Journal of the ACM,* Vol. 18, No. 1, 80–93, January 1971.

[2] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer, *IBM System Journal,* 5 (2) 78–101, 1966.

[3] Azer Bestavros, Robert L. Carter, Mark E. Crovella, Carlos R. Cunha, Abdelsalam Heddaya, Sulaiman A. Mirdad "Application-Level Document Caching in the Internet," *Proceeedings of the Second Workshop on Services in Distributed and Networked Environments (SDNE '95),* June 1995.

[4] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, "Introduction to Algorithms," *The MIT Press,* Cambridge Massachussetts, 1990.

[5] Steve Glassman, "A Caching Relay for the World Wide Web," *Computer Networks and ISDN Systems,* Vol 27 Number 2, November 1994.

[6] R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM System Journal,* 5 (2), 78–117, 1970.

[7] Christos H. Papadimitriou, S. Ramanathan, P. Venkat Rangan, "Information Caching for Delivery of Personalized Video Programs over Home Entertainment Channels," *Proceedings of The IEEE International Conference on Multimedia Computing and Systems,* Boston, MA, May 1994.

[8] S. Ramanathan, P. Venkat Rangan, "Architectures for Personalized Multimedia," *IEEE Multimedia,* Vol. 1, No. 1, Spring 1994.

[9] Bernard Sklar, "Digital Communication: Fundamentals and Applications," *Prentice Hall,* New Jersey, 1988.

[10] Allan J. Smith, "Bibliography on Paging and Related Topics," *Operating Systems Review,* vol. 12, 39–56, Oct. 1978.

[11] Allan J. Smith, "Second Bibliography for Cache Memories," *Computer Architecture News,* Vol. 19, No. 4, June 1991.

[12] H. C. Tijms, "Stochastic Models: An Algorithmic Approach," *John Wiley & Sons,* New York, 1994.