# Principles of Optimal Page Replacement

ALFRED V. AHO

*Bell Telephone Laboratories, Inc., Murray Hill, New Jersey*

AND

PETER J. DENNING AND JEFFREY D. ULLMAN

*Princeton University, Princeton, New Jersey*

ABSTRACT. A formal model is presented for paging algorithms under $l$-order nonstationary assumptions about program behavior. When processing a program under paging in a given memory, a given paging policy generates a certain (expected) number of page calls, i.e., its "cost." Under usual assumptions about memory system organization, minimum cost is always achieved by a demand paging algorithm. The minimum cost for $l$-order program behavior assumptions is expressed as a dynamic programming problem whose solution yields an optimal replacement algorithm. Solutions are exhibited in several 0-order cases of interest. Paging algorithms that implement and approximate the minimum cost are discussed.

KEY WORDS AND PHRASES: paging algorithms, demand paging algorithms, demand paging, replacement algorithms, program models, program behavior, virtual memory, storage allocation, dynamic storage allocation

CR CATEGORIES: 4.3, 5.2

## 1. *Introduction*

Since the earliest days of electronic computing, machines with very large over-all memory capacity have used memory systems consisting of at least two levels. Information must reside in "main memory" to be accessible by a processor, and it may otherwise reside in "auxiliary memory." Older systems gave the programmer control over the decisions that moved information between main and auxiliary memory. But many newer systems, using virtual memory [1–4], provide a programmable address space many times larger than the available main memory; these systems reserve for themselves control over which parts of a programmer's address space shall reside in main memory, and when information shall be moved between main and auxiliary memory. Hereafter, the term "memory" will be used for "main memory."

Many virtual memory systems use *paging*, an addressing scheme that organizes memory into fixed-size blocks of locations called *page frames*, and organizes address space into matching-size blocks of addresses called *pages*. Memory is regarded as a pool of anonymous page frames, any page being placeable in any page frame, and the page is the unit of information transmission between main and auxiliary memory.

Suppose the set $N = \{1, \cdots, n\}$ denotes a program's pages, the set

$$M = \{1, \cdots, m\}$$

Work reported herein was supported in part by National Aeronautics and Space Administration Grant NGR-31-001-170.

denotes the memory page frames, and $1 \leq m \leq n$. At each moment of time there is a function $f: N \rightarrow M$ given by

$$f(i) = \begin{cases} j & \text{if page } i \text{ resides in frame } j, \\ \text{undefined} & \text{otherwise,} \end{cases}$$

which is called the "page map." If the page size is $z$, a valid virtual address is an integer $\alpha$, $0 \leq \alpha < nz$. Similarly, a valid memory address is an integer $\beta$ $(0 \leq \beta < mz)$. When presented with virtual address $\alpha$, the address translation hardware finds $(i, w)$ such that $\alpha = (i - 1)z + w$ and $0 \leq w < z$, and then generates memory address $\beta = [f(i) - 1]z + w$ if $f(i)$ is defined and a "page fault" if $f(i)$ is undefined. A page fault interrupts program execution for one "page wait time," until the system can secure the missing page from auxiliary memory, load it into an available frame of main memory, and update the page map to account for the change in memory contents.

A *paging policy*—which specifies how to use the paging mechanism—comprises three subpolicies. The *replacement policy* is used to select a page to be removed from memory. The *placement policy* is used to select, from among available frames, a frame in which to place an incoming page. The *fetch policy* is used to determine when a page shall enter memory. If pages are brought into memory when and only when faults for them occur, the fetch policy is called a *demand policy*; otherwise it is called a *nondemand*, or *prepaging*, *policy*. A *paging algorithm* is an implementation of a paging policy.

The criterion usually taken for paging algorithm optimality is minimization of the total real time required to execute a program. Because it greatly simplifies the analysis without significantly affecting the generality of our treatment, we take as our criterion of paging algorithm optimality the minimization of the aggregate page wait time accumulated while executing a program. If a program's page wait intervals and execution intervals are disjoint (e.g., under demand fetch policies), then the two criteria are equivalent. If the average time between page faults is much less than the average page wait time (e.g., with rotating auxiliary memories), then the second criterion closely approximates the first.

The first virtual memory computer (the Atlas [1]), and a great many virtual memory computers since, have used "demand paging," a paging policy that uses a demand fetch policy and keeps memory as full as possible. Under demand paging, the placement policy is trivial: if memory is not full, one places the incoming page arbitrarily in any unused frame; or if memory is full, one first uses the replacement policy to free a frame, then places there the incoming page. This leaves the replacement policy as the only component of a demand paging policy subject to manipulation, and thus the study of optimal demand paging policies reduces to a study of optimal replacement policies. Since each page wait interval is initiated by a page fault under demand paging, a demand algorithm will be optimal if and only if it minimizes the number of page faults.

We limit our attention to optimal demand policies for two reasons. First we shall prove that, with certain constraints on memory system organization, an optimal paging policy must be a demand policy. Second, a great many systems for which a nondemand policy would in theory be optimal are committed by their implementations to using demand paging only.

Beginning with the first page replacement policy [1], there has been a widely accepted "informal principle of optimality" for replacement policies: *the page to be replaced is that which has the longest expected time until next reference.* The variety of policies that have been proposed over the years results from the various assumptions about program behavior used to determine the expected time until next reference. Experimental evidence [5, 6] suggests that this principle is a very good heuristic for reasonable assumptions about program behavior.

Despite its wide acceptance and intuitive appeal, it is easy to find counterexamples to the informal principle of optimality for arbitrary assumptions about program behavior.[1] In the following we shall define a hierarchy of $l$-order nonstationary Markov models for program behavior, then develop a dynamic programming problem whose solution is the optimal replacement policy. We are able to solve the dynamic programming problem for the optimal policy in several 0-order cases of interest, and to show in these cases that the optimal policy implements the informal principle of optimality.

There are few published papers offering experimental studies [5–9] or formal models [10–16] of program and replacement algorithm behavior. Two papers have proposed conjectured optimal replacement policies: the Belady algorithm [5] presumes exact knowledge of future page references, and the Denning–Chen-Shedler algorithm [13] presumes 0-order stationary Markov statistics for future page references. These and other algorithms fall in the class to be studied below.

## 2. *Page Algorithms*

As before, we suppose $N = \{1, \cdots, n\}$ are the pages of a given $n$-page program $M = \{1, \cdots, m\}$ are the page frames of memory, and $1 \leq m \leq n$. The set $N^k$ comprises all strings of length $k$ over $N$, where $k \geq 0$. We may model the page-referencing activity of a program by its *reference string*, which is an element $\omega = r_1 \cdots r_t \cdots r_T$ of $N^T$ for some $T$; if $r_t = x$, we understand that the program references page $x$ at the $t$th reference. By regarding the nonnegative integers as "time instants," we may say that $r_t$ is the "page referenced at time $t$."

We call $S \subseteq N$ a *memory state*. Let $\mathfrak{M}_m$ be the set $\{S \mid S \subseteq N \text{ and } |S| \leq m\}$ where $|X|$ denotes the number of elements in set $X$.

*Definition 1.* Let $M$ and $N$ be given. A *paging algorithm for $M$ and $N$* is a system $A = (Q, q_0, g)$ in which

(1) $Q$ is a set of *control states* of the algorithm.

(2) $q_0$ in $Q$ is the *initial control state.*

(3) $g: \mathfrak{M}_m \times Q \times N \rightarrow \mathfrak{M}_m \times Q$ is the *allocation map.*

The allocation map has the property that $x$ is in $S'$ whenever $g(S, q, x) = (S', q')$.

Note that a paging algorithm is nothing more than a state machine with state set $\mathfrak{M}_m \times Q$, inputs $N$, and transition function $g$. A pair $(S, q)$ in $\mathfrak{M}_m \times Q$ is called a *configuration* of $A$.

*Definition 2.* Suppose memory state $S_0$ and reference string $\omega = r_1 \cdots r_t \cdots r_T$ are given. Paging algorithm $A$ *processes* $\omega$ from initial state $S_0$ by generating the sequence of configurations $\{(S_t, q_t)\}_{t=0}^{T}$ such that

$$(S_t, q_t) = g(S_{t-1}, q_{t-1}, r_t) \qquad (1 \leq t \leq T).$$

[1] We are grateful to John Pomeranz, University of Chicago, for pointing one out to us.

If $r_t \notin S_{t-1}$, a *fault* occurs at time $t$. Each memory state is related to its predecessor by

$$S_t = S_{t-1} \cup X_t - Y_t$$

where $X_t \subseteq N - S_{t-1}$ and $Y_t \subseteq S_{t-1}$. The pages $X_t$ are those *placed* in memory, and $Y_t$ are those *replaced* from memory.

*Definition 3* (Notation).

$S_t$ denotes the memory state at time $t$.

$S_t^m$ denotes a memory state of size $m$ (i.e. $|S_t^m| = m$).

$S + x$ denotes the set $S \cup \{x\}$, where $x \notin S$.

$S - x$ denotes the set $S - \{x\}$, where $x \in S$.

*Definition 4.* The paging algorithm $A$ is a *demand paging algorithm* if its allocation map is of the form

$$g(S, q, x) = \begin{cases} (S, q') & \text{if } x \in S, \\ (S + x, q') & \text{if } x \notin S, |S| < m, \\ (S + x - y, q') & \text{if } x \notin S, |S| = m, y \in S. \end{cases}$$

If $|S_0| < m$, then in most cases under demand paging the memory eventually fills and remains full thereafter.

*Example 1.* Two common demand paging algorithms are FIFO (first-in-first-out) and LRU (least-recently-used) [5, 9, 12]. For both FIFO and LRU, the algorithm control states are $Q = N^m$, a typical such state being $q = (y_1, \cdots, y_m)$. When $|S| = m$, the allocation maps are

$$g_{\text{FIFO}}(S, q, x) = \begin{cases} (S, q) & \text{if } x \in S, \\ (S + x - y_m, q') & \text{if } x \notin S, \end{cases}$$

$$g_{\text{LRU}}(S, q, x) = \begin{cases} (S, q'') & \text{if } x \in S, \\ (S + x - y_m, q') & \text{if } x \notin S, \end{cases}$$

where $q' = (x, y_1, \cdots, y_{m-1})$, and if $x = y_k$, then

$$q'' = (x, y_1, \cdots, y_{k-1}, y_{k+1}, \cdots, y_m).$$

Let $\tau(x, t)$ be the time to the next reference to page $x$ time $t$; if $x$ is not referenced again after time $t$, take $\tau(x, t) = T - t + 1$.

*Example 2.* Belady's proposed optimal algorithm, which we denote $B_0$, assumes that the reference string $\omega = r_1 \cdots r_t \cdots r_T$ is known, and replaces that page $y$ with the longest time to next reference, $\tau(y, t)$. The algorithm control states are the times $t$ $(0 \leq t \leq T)$. For $|S| = m$, the allocation map is specified by

$$g_{B_0}(S, t, r_{t+1}) = \begin{cases} (S, t + 1) & \text{if } r_{t+1} \in S, \\ (S + r_{t+1} - y, t + 1) & \text{if } r_{t+1} \notin S, \end{cases}$$

where $y$ is a member of $S$ such that $\tau(y, t) \geq \tau(x, t)$ for all $x$ in $S$.

*Example 3.* The Denning–Chen–Shedler proposed optimal algorithm [13], which we denote by $A_0$, replaces the page with the longest expected time until next reference. Their assumption about program behavior is that $E[\tau(x, t)]$ is fixed for all $t > 0$. Suppose the pages have been numbered such that $E[\tau(x, t)] \leq E[\tau(x + 1, t)]$

for $1 \leq x < n$. The algorithm has one control state $q_0$ and allocation map

$$g_{A_0}(S, q_0, x) = \begin{cases} (S, q_0) & \text{if } x \in S, \\ (S + x - y, q_0) & \text{if } x \notin S, \end{cases}$$

where $y$ is the largest integer in $S$, and $|S| = m$.

### 3. *Cost of Algorithms*

By Definition 2, $|X_t|$ pages enter memory and $|Y_t|$ pages exit at time $t$. In calculating the cost of these transactions, one usually ignores the cost of removing the pages $Y_t$ for one or both of these reasons: (1) it is sometimes possible to make the operations of removing $Y_t$ concurrent with those of placing $X_t$; (2) over a long period of time, the number of pages removed tends to be some fixed fraction of the number of pages placed.

*Definition 5.* Let $h(k)$ denote the cost of an operation that places $k \geq 1$ pages in memory, where $h(k) \geq h(1) = 1$. The *cost* for processing $\omega = r_1 \cdots r_t \cdots r_T$ from initial memory state $S$ under paging algorithm $A$ is

$$C(A, S, \omega) = \sum_{t=1}^{T} h(|X_t|).$$

If $A$ is a demand paging algorithm, $|X_t| \leq 1$ for $1 \leq t \leq T$, and the cost is simply

$$C(A, S, \omega) = \sum_{t=1}^{T} |X_t|.$$

In case $\omega$ is a random variable over $N^k$, we define the *expected cost*

$$C_k(A, S) = \sum_{\omega \in N^k} \Pr[\omega] C(A, S, \omega)$$

where Pr [ ] denotes a probability function on $N^k$.

A one-page transport operation takes time $T_w + T$, where $T_w$ is a "waiting time" and $T$ is a "transmission time." Depending on the organization of auxiliary memory, a $k$-page transport operation may take time $k(T_w + T)$ or time $T_w' + kT$, $T_w' \geq T_w$. The latter expression would hold for systems in which most transfers occur between random-access and sequential-access memories (e.g. "core-to-drum"). The former expression would hold for systems in which most transfers occur between high-speed and low-speed random-access memories (e.g. "core-to-core"); memory systems of this type are increasingly important [4]. Thus, in the latter case we would assign $h(k) < k$, and in the former case $h(k) = k$ for $k \geq 1$.

Theorem 1 establishes that if $h(k) \geq k$, an optimal paging algorithm must be a demand algorithm. This result is a straightforward generalization of one in [15].

THEOREM 1. *Let $A = (Q, q_0, g)$ be a paging algorithm and suppose $h(k) \geq k$ for $k \geq 1$ and $h(1) = 1$. There exists a demand algorithm $A'$ such that $C(A', S_0, \omega) \leq C(A, S_0, \omega)$ for all $S_0$ and $\omega$.*

PROOF. Assume $A$ is a nondemand algorithm. We construct a demand algorithm $A'$ which keeps track of pages placed and replaced by $A$ in its control state. However, $A'$ places only those pages $A$ would have been required to place. Formally, let $A' = (Q \times 2^N \times 2^N, q_0', g')$. If $A$ is in configuration $(S, q)$, then $A'$ will be in configuration $(S', (q, P, R))$ where $P$ contains the pages placed by $A$ but not yet by $A'$

(i.e., deferred placements), $R$ contains pages replaced by $A$ but not yet by $A'$ (i.e., deferred replacements), and $S' = S - P \cup R$. Suppose $g(S, q, i) = (S \cup X - Y, q')$ where $X \cap Y = \varnothing$. Then for $S' = S - P \cup R$, $P' = P - Y \cup X$, and $R' = R - X \cup Y$,

$$g(S', (q, P, R), x) = \begin{cases} (S', (q', P', R')) & \text{if } x \in S', \\ (S' + x, (q', P' - x, R')) & \text{if } x \notin S', \ |S'| < m, \\ (S' + x - y, (q', P' - x, R' - y)) & \text{if } x \notin S', \ |S'| = m, \ y \in S' \\ & \quad (\text{any } y \text{ will do, e.g. least integer in } S'). \end{cases}$$

By construction, every page placed by $A'$ must also have been placed by $A$. Therefore $C(A, S_0, \omega) = \sum_t h(|X_t|) \geq \sum_t |X_t| \geq \sum_t |X_t'| = C(A', S_0, \omega)$.

Even though the case $h(k) < k$ occurs frequently in current practice, there is no simple condition on $h$ that determines when a demand algorithm will have lower cost than a prepaging algorithm, because the condition will depend on the probability assignment on reference strings [4, 15]. Therefore, an optimal demand algorithm will always be an optimal paging algorithm if the condition of Theorem 1 holds, and will sometimes be an optimal paging algorithm otherwise. Hereafter we limit attention to demand algorithms, and we shall use the term "algorithm" for "demand algorithm."

When algorithm $A$ and reference string $\omega$ are given, there is a "best" starting state. If $|S_0| < m$, we may construct the following state $S_0'$. Let $k = m - |S_0|$ and let $T_0$ be the $k$ distinct pages in $N - S_0$ to be referenced first in $\omega$. Then $A$ started in $S_0' = S_0 \cup T_0$ will generate $k$ fewer faults than $A$ started in $S_0$. Repeating this argument with $S_0 = \varnothing$ gives the "best" starting state. In case $\omega$ is a random variable, $S_0'$ is the set of $m$ pages most likely to be referenced first. Thus, without loss of generality, we may suppose that the initial state $S_0$ contains $m$ pages.

## 4. *Principle of Optimality*

A general formulation of the mechanism by which programs generate reference strings is the following $l$-order nonstationary Markov process.

*Definition 6.* A *program* is a system $P = (N, U, u_0, f, p)$ in which $N$ is a set of pages, $U$ is a set of "program states" including initial state $u_0$, $f: N \times U \to U$ is the state transition map, and $p$ is a probability function; $p(x, u, t)$ is the probability that a reference to page $x$ is generated at time $t$ given that $P$ is currently in state $u$. Thus, for all $u$ in $U$ and $t > 0$, $\sum_{x \in N} p(x, u, t) = 1$. The program $P$ generates a reference string $r_1 \cdots r_t \cdots r_T$ as follows. If $u_{t-1} = u$, then $r_t$ has the value $x$ with probability $p(x, u, t)$ and $u_t = f(r_t, u_{t-1})$. The program is *l-order* if $|U| = l + 1$. It is *stationary* if $p$ is independent of $t$.

*Example 4.* An $n$-page program in which $r_t$ depends only on $r_{t-1}$ for $t > 1$ is an $n$-order program by the foregoing definition. The program states are $U = N \cup \{u_0\}$ and the transition map is defined by $f(x, u) = x$ for $x$ in $N$ and $u$ in $U$. If $p_{ij}(t) = \Pr[r_t = j \mid r_{t-1} = i]$, then take $p(x, u, t) = p_{ux}(t)$ in the definition.

Generalizing, if $P$ is an $n$-page program where $r_t$ depends on $r_{t-k} \cdots r_{t-1}$, then $U = N^k \cup N^{k-1} \cup \cdots \cup N^0$; a typical element of $U$ represents the most recent $k$

references generated by $P$. Thus $f(x, a_1 \cdots a_i)$ is $xa_1 \cdots a_i$ if $i < k$, and $xa_1 \cdots a_{k-1}$ if $i = k$.

We shall construct a recursive definition of the minimum achievable cost of processing, under demand paging, a reference string generated by an $l$-order program. Suppose $\omega = r_1 \cdots r_t \cdots r_{t+k}$, $u_t = u$, and $S_t = S$. The minimum cost for processing the $k$ references beyond time $t$ is denoted by $C_k(S, u, t)$ and is defined by

$$C_0(S, u, t) = 0,$$

$$C_k(S, u, t) = \sum_{x \in N} p(x, u, t+1) \times \begin{cases} C_{k-1}(S, f(x, u), t+1) & \text{if } x \in S, \\ 1 + \min_{z \in S} C_{k-1}(S + x - z, f(x, u), t+1) & \text{if } x \notin S. \end{cases} \tag{1}$$

*Definition 7.* An algorithm $A$ is said to be *$l$-optimal* if for all $T$ and $S$, $C_T(A, S) = C_T(S, u_0, 0)$ whenever the probability of reference strings of length $T$ is determined by an $l$-order program $P = (N, U, u_0, f, p)$. We use $A_l$ to refer to an $l$-optimal algorithm.

In general, $A_l$ may not be feasible to implement. Not only would it require information about $p(x, u, t)$ and assumptions about the length of the reference string (which presumably is unknown in advance of execution), but it may require an immensely complex computation to solve (1). The case $l = 0$ is one in which solutions to (1) may be implemented by simple algorithms.

### 5. *0-Order Solutions*

Suppose $P$ is truly an $l$-order program, i.e. it is not equivalent to any program of smaller order. Suppose further that $A$ is an optimal algorithm for $P$. On intuitive grounds, $A$ must use its control states to estimate the program states of $P$; in other words, all or most of the program states must be reflected in those of the paging algorithm. Although it appears very difficult to determine when two distinct program states may be reflected in one algorithm control state, there are easy examples of cases in which this may be done (e.g., two program states having identical successors associate different probabilities with pages, but these probabilities differ in ways so small that they fail to affect the optimal paging strategy).

There are several reasons why we have chosen to limit attention to 0-order algorithms in the remainder of this paper. First, there is little agreement on the minimum complexity required for a "correct" program model, and there is some merit to the argument that the most unstructured model may yield insight into paging algorithm behavior for the largest class of programs. Second, a simple model corresponds to the universal desire for easily implemented algorithms. Third, even this problem appears intractable in the general time-varying case: we are able to prove the optimality of a 0-order algorithm only when the probabilities—even though time-varying—maintain the relative likelihood of pages being referenced. At this point we are able to give only approximate extensions to the general 0-order case. Thus, because of the intrinsic difficulty of the problem, we believe that the simplest program model is a good starting point for the formal investigation of paging algorithm behavior.

In the 0-order case, a program $P = (N, U, u_0, f, p)$ has just one program state

i.e. $U = \{u_0\}$. We shall write $p(x, t)$ for $p(x, u_0, t)$, so that eqs. (1) reduce to $C_0(S, t) = 0$,

$$C_k(S, t) = \sum_{x \in N} p(x, t+1) \cdot \begin{cases} C_{k-1}(S, t+1) & \text{if } x \in S, \\ 1 + \min_{s \in S} C_{k-1}(S + x - z, t+1) & \text{if } x \notin S. \end{cases} \quad (2)$$

*Definition 8* (Notational Summary). For a 0-order reference string $r_1 \cdots r_t \cdots r_T$,
$p(x, t)$ denotes $Pr[r_t = x]$ for $x$ in $N$;
$\tau(x, t)$ denotes the time to next reference of $x$ after time $t$, and $\tau(x, t) = T - t + 1$ if $x$ is not referenced again;
$C_k(S, t)$ denotes the minimum demand paging cost of processing $r_t \cdots r_{t+k}$, when $t + k \leq T$ and $S_t = S$.

The following discussion is motivated by the informal observation that pages with larger values of $p(x, t)$ are "more useful," having smaller expected times until reuse. In other words, $p(x, t) < p(y, t)$ suggests that $\tau(x, t) > \tau(y, t)$ if $p(x, t)$ and $p(y, t)$ do not vary rapidly with time. The informal principle of optimality then advises us to replace $s$ whenever $p(s, t)$ is the smallest of the $p(x, t)$ in memory. We consider first a simple case—"almost" stationary probabilities—and obtain the solution to (2); later we discuss generalizations to nonstationary cases.

*A. Almost Stationary Case*

By the almost stationary case, we mean that the probabilities maintain their relative order with respect to time. That is, if $p(x, t) \geq p(y, t)$, then $p(x, t + t') \geq p(y, t + t')$ for all $t, t' \geq 0$. In this case, we are able to define a relation $<$ on $N$ and show that, if $s$ is the smallest element of $S$ according to $<$, then

$$C_k(S - s, t) = \min_{x \in S} C_k(S - x, t).$$

This relation will provide a reasonable way of implementing the solution to (2), since then the optimal algorithm will always replace the lowest ranked page in memory. Even though this result is quite intuitive, the proof turns out to be somewhat lengthy.

*Definition 9.* A *stationary ranking relation* is a binary relation $<$ on $N$ such that if $x < y$, then $p(x, t) \leq p(y, t)$ for all $t > 0$. The notation $x \leq y$ means: $x < y$ or $x = y$. The notation $s = \min S$ means: $s \in S$ and $s \leq x$ for all $x \in S$.

LEMMA 1. *For some $t > 0$ and for all $S' \subset N$, suppose that $x < y$ implies $C_k(S' + x, t) \geq C_k(S' + y, t)$. Then $s = \min S$ implies that $C_k(S - s, t) = \min_{z \in S} C_k(S - z, t)$ for all $S \subset N$.*

PROOF. It suffices to show that for all $z \neq s$ in $S$, $C_k(S - s, t) \leq C_k(S - z, t)$. If we assume $s < z$, the result follows by letting $x = s$, $y = z$, and $S' = S - s - z$.

LEMMA 2. *Suppose $<$ is a stationary ranking of $N$. Then*

$$x < y \Rightarrow 1 \geq C_k(S + x, t - 1) - C_k(S + y, t - 1) \geq 0$$

*where $x, y$ are not in $S$ and $t \geq 1$. (We use $t - 1$ as a parameter instead of $t$ to avoid many occurrences of expression $t + 1$.)*

PROOF. Applying (2), we may write

$$\Delta = C_k(S + x, t - 1) - C_k(S + y, t - 1) = \sum_{i \in S} p(i, t)\Delta_1 + \sum_{i \in S+x+y} p(i, t)\Delta_2 + p(x, t)(\Delta_3 + \Delta_4) + [p(y, t) - p(x, t)]\Delta_4$$

where

$$\Delta_1 = C_{k-1}(S + x, t) - C_{k-1}(S + y, t),$$
$$\Delta_2 = \min_{s \in S+x} C_{k-1}(S + x + i - z, t) - \min_{s' \in S+y} C_{k-1}(S + y + i - z', t),$$
$$\Delta_3 = C_{k-1}(S + x, t) - \min_{s' \in S+y} C_{k-1}(S + y + x - z', t) - 1,$$
$$\Delta_4 = \min_{s \in S+x} C_{k-1}(S + x + y - z, t) - C_{k-1}(S + y, t) + 1.$$

Note that instances of $p(x, t)\Delta_4$ cancel in the above. Since $C_0(S + x, t) = C_0(S + y, t) = 0$, Lemma 2 holds for $k = 0$. Suppose by induction it holds for $0 \leq k < k'$; we shall show it holds for $k = k'$. Noting that letting $s = x$, $y \notin S$ min $S$, and applying Lemma 1 to the inductive hypothesis, we find there are three cases to consider:

(i) $s < x < y \Rightarrow z = s$, $z' = s$,
(ii) $x < s < y \Rightarrow z = x$, $z' = s$,
(iii) $x < y < s \Rightarrow z = x$, $z' = y$.

Clearly $1 \geq \Delta_1 \geq 0$ by the inductive hypothesis. For the three cases, the forms of $\Delta_2$ are

(i) $\Delta_2 = C_{k-1}(S + x + i - s, t) - C_{k-1}(S + y + i - s, t)$
(ii) $\Delta_2 = C_{k-1}(S + i, t) - C_{k-1}(S + y + i - s, t)$,
(iii) $\Delta_2 = C_{k-1}(S + i, t) - C_{k-1}(S + i, t) = 0$.

In case (i) the inductive hypothesis applies directly. In case (ii) the inductive hypothesis also applies noting that $C_{k-1}(S + i, t) - C_{k-1}(S + y + i - s, t) = C_{k-1}(T + s, t) - C_{k-1}(T + y, t)$ where $T = S + i - s$. Thus, in all three cases, $1 \geq \Delta_2 \geq 0$.

The three possible forms for $(\Delta_3 + \Delta_4)$ are

(i) $\Delta_3 + \Delta_4 = C_{k-1}(S + x, t) - C_{k-1}(S + y + x - s, t)$
$\quad + C_{k-1}(S + x + y - s, t) - C_{k-1}(S + y, t)$,
$\quad = C_{k-1}(S + x, t) - C_{k-1}(S + y, t)$;

(ii) $\Delta_3 + \Delta_4 = C_{k-1}(S + x, t) - C_{k-1}(S + y + x - s, t) + C_{k-1}(S + y, t)$
$\quad - C_{k-1}(S + y, t)$
$\quad = C_{k-1}(S + x, t) - C_{k-1}(S + y + x - s, t)$;

(iii) $\Delta_3 + \Delta_4 = C_{k-1}(S + x, t) - C_{k-1}(S + x, t) + C_{k-1}(S + y, t)$
$\quad - C_{k-1}(S + y, t)$
$\quad = 0$.

Applying the inductive hypothesis where needed, it can be shown that in each case $1 \geq \Delta_3 + \Delta_4 \geq 0$. It is convenient to express $\Delta_4$ in the form $\Delta_4 = 1 - \Delta_5$, where

$$\Delta_5 = C_{k-1}(S + y, t) - \min_{s \in S+x} C_{k-1}(S + y + x - z, t).$$

There are two cases to consider:

$$s < x \Rightarrow z = s \Rightarrow \Delta_5 = C_{k-1}(S + y, t) - C_{k-1}(S + x + y - s, t),$$
$$x < s \Rightarrow z = x \Rightarrow \Delta_5 = C_{k-1}(S + y, t) - C_{k-1}(S + y, t) = 0.$$

In both cases, $1 \geq \Delta_5 \geq 0$ and hence $1 \geq \Delta_4 \geq 0$. Using the upper and lower bounds on the $\Delta_i$ in the equation for $\Delta$ and noting that $p(y, t) \geq p(x, t)$, we find $1 \geq \Delta \geq 0$, as required.

THEOREM 2. *If program $P$ is 0-order and has a stationary ranking $<$ on $N$, then*

*the optimal algorithm $A_o$ has the allocation map $g_{A_o}: \mathfrak{M}_m \times N \to \mathfrak{M}_m$ given by*

$$g_{A_o}(S, x) = \begin{cases} S & \text{if } x \in S, \\ S + x - s & \text{if } x \notin S, \end{cases}$$

*where $s = \min S$ and $|S| = m$.*

PROOF. By Lemmas 1 and 2, $A_o$ makes exactly the decisions implied by (2).

Suppose there is exactly one ranking relation on $N$ [i.e., $p(x, t) \neq p(y, t)$ for all $x, y \in N$, and $t > 0$]; the proof of Lemma 2 then shows that $C_k(S + x, t) - C_k(S + y, t) \neq 0$ whenever $x \neq y$. If algorithm $A$ makes at least one decision differently from $A_o$, then $C_k(A, S, t) > C_k(A_o, S, t)$. Similarly, if $p(x, t) = p(y, t)$ for $x, y \in N$, and $t > 0$ (in which case every permutation of $N$ defines a valid ranking relation), the proof of Lemma 2 then shows that $C_k(S + x, t) - C_k(S + y, t) = 0$. Then $C_k(A, S, t) = C_k(A_o, S, t)$ for every algorithm $A$.

*B. Properties of the Optimal Algorithm*

THEOREM 3. *Suppose $<$ is a stationary ranking of $N$, and the corresponding algorithm $A_o$ with $|M| = m$ generates states $\{S_t^m\}_{t \geq 0}$ for some reference string $\omega = r_1 \cdots r_T$. If $S_o^m \subset S_o^{m+1}$, then*

$$S_t^m \subset S_t^{m+1}, \qquad t \geq 0,\ 1 \leq m < n.$$

PROOF. Suppose Lemma 2 holds for $0 \leq t < t'$; we show it holds for $t = t'$. Let $S_{t-1}^m = S$, $S_{t-1}^{m+1} = S + x$, $s = \min S$, and $s' = \min(s, x)$. If $r_t \in S$, then $S_t^m = S \subset S + x = S_t^{m+1}$. If $r_t \notin S + x$ and $s' = s$, then $S_t^m = S + r_t - s \subset S + x + r_t - s = S_t^{m+1}$; or if $s' = x$, then $S_t^m = S + r_t - s \subset S + r_t = S_t^{m+1}$. If $r_t = x$, then $S_t^m = S + x - s \subset S + x = S_t^{m+1}$.

Theorem 3 shows that the memory states $S_t^m$ satisfy the "inclusion property" (this property is used in [15] to analyze a class of "stack algorithms"). This implies that, if there is a fault at time $t$ in a memory of given size, there must simultaneously be one in every memory of smaller size.

THEOREM 4. *Suppose $|S| = m$ $(t \geq 0)$ and $p(i, t) > 0$ for all $i \in N$ and $t > 0$. Then*

$$C_k(S + x, t) < C_k(S, t) \qquad (1 \leq m \leq n,\ k > 0).$$

In physical situations, it would be true that $p(i, t) > 0$ for $t > 0$. Theorem 4 then guarantees that an increase in memory always results in a decrease in cost. This is not so for arbitrary algorithms: examples are known in which the FIFO algorithm may exhibit as much as twice the cost for a memory size increase of one page [15, 16].

*Definition 9.* Define the set $L^m$ to be the $m - 1$ highest-ranked pages in $N$ $(m \geq 1)$. For starting state $S_0^m$, $A_o$ is said to be in *steady state* whenever $S_t^m \supset L^m$. The *settling time* $T(S)$ is the expected time required for $A_o$ to enter steady state when started with initial memory state $S$.

Clearly, if $S \supset L^m$, then $T(S) = 0$; such an $S$ is a "good" starting state. [We shall determine a bound on $T(S)$ below.] Although $T(S)$ need not, in general, be short, the cost of not starting in a "good" starting state is low.

LEMMA 3. *Let $S_1$ and $S_2$ be sets of $m$ pages each, and suppose we can write $S_1 = S \cup S_1'$, $S_2 = S \cup S_2'$, where $|S_1'| = |S_2'| = h$ and $S_1' \cap S_2' = \varnothing$. Then $|C_k(S_1, t) - C_k(S_2, t)| \leq h$.*

PROOF. We proceed by induction on $h$. The basis, $h = 0$, is trivial. Suppose the result holds for $h - 1$, and let $x$ be in $S_1'$ and $y$ in $S_2'$. Consider $S_3 = S_1 - x + y$. By the inductive hypothesis,

$$| C_k(S_3, t) - C_k(S_2, t) | \leq h - 1.$$

By Lemma 2, $| C_k(S_1, t) - C_k(S_3, t) | \leq 1$. The result follows immediately.

We direct attention now to the "completely" stationary case, in which the page reference probabilities are independent of time, so that we may write $p(x, t) = p_x$ for all $x$ in $N$ and $t \geq 0$, and the cost $C_k(S, t)$ can now be written as $C_k(S)$. In this case an expression for the limiting cost per reference can be obtained [13]. Moreover, $A_0$ is now the algorithm given in Example 3.

THEOREM 5. *Suppose the 0-order page reference probabilities are stationary. Under $A_0$ the expected cost per reference is*

$$C'(S) = \lim_{k \to \infty} \frac{C_k(S)}{k} = B - \frac{1}{B}\left(\sum_{i=m}^{n} p_i^2\right)$$

*where $| S | = m$, $N = \{1, \cdots, n\}$ such that $p_1 \geq \cdots \geq p_n$, and $B = \sum_{i=m}^{n} p_i$.*

PROOF. The memory state sequence $S_0 S_1 \cdots S_k$, $k \geq 0$, generated by $A_0$ is a sequence of random variables whose values lie in $\mathfrak{M}_m$. Let $a_x(t) = \Pr[x \notin S_t]$ for $t \geq 0$ and $x \in N$. Then the expected cost $C_k(S)$ can be expressed as follows:

$$\begin{aligned} C_k(S_0) &= \sum_{t=1}^{k} \Pr[r_t \notin S_{t-1}] \\ &= \sum_{t=1}^{k} \sum_{x \in N} p_x a_x(t-1). \end{aligned}$$

By Lemma 3, when calculating the limiting cost per reference, it is sufficient to consider a starting memory state $S_0$ such that $S_0 \supset L^m$ and the $x$ in $S_0 - L^m$ is chosen with probability $p_x$. Then for $t \geq 0$,

$$a_x(t) = \begin{cases} 0 & \text{if } 1 \leq x < m, \\ 1 - p_x/B & \text{if } m \leq x < n, \end{cases}$$

where

$$B = \sum_{i=m}^{n} p_i$$

and

$$C_k(S_0) = k\left[B - \frac{1}{B}\left(\sum_{i=m}^{n} p_i^2\right)\right].$$

By Lemma 3, the limiting cost per reference for $A_0$ beginning in any starting memory state $S$ is the limiting cost per reference for $A_0$ in steady state. Thus

$$\begin{aligned} C'(S) &= \lim_{k \to \infty} \frac{C_k(S)}{k} \\ &= B - \frac{1}{B}\left(\sum_{i=m}^{n} p_i^2\right). \end{aligned}$$

Starting from an arbitrary initial memory state $S_0$, we observe that the limiting, or steady state, condition begins to hold for the smallest $t$ such that $S_t^m \supset L^m$.

Letting $S = L^m - S_0$, an upper bound on the settling time $T(S_0)$ is

$$T(S_0) \leq \sum_{i \in S} \frac{1}{p_i}. \tag{3}$$

Expression (3) follows from the observation that the expected time between references to page $i$ is $1/p_i$, since all references are statistically independent in the stationary 0-order case.

*C. Nonstationary 0-Order Case*

With simple modifications, the technique used in Lemmas 1 and 2 can be used to show that Belady's algorithm (Example 2) is optimal when the reference string is known. Since the detailed proof of this has been given in [15], we shall only outline the changes. Let $\omega = r_1 \cdots r_t \cdots r_T$ and consider the following nonstationary ranking relation $<_t$ on $N$. If $x <_t y$, then $y$ appears before $x$ in the sequence $r_t, r_{t+1}, \cdots, r_T$ or $x$ does not appear at all. Define $p(x, t) = 1$ if $r_t = x$ and 0 otherwise; note that $0 = p(x, t) \leq p(y, t) \leq 1$ if $x <_t y$, and note that $x <_t y$ implies $x <_{t+1} y$ if $r_t \neq y$. This ranking relation, together with these conditions on the $p(i, t)$, are sufficient to make Lemma 2 go through.

Our investigations appear to indicate that solution of (2) using the ranking-relation technique holds only in the almost stationary case and in the special nonstationary situation mentoned above. (This does not, however, preclude the existence of other $A_0$ algorithms based on the informal principle of optimality.) Therefore, some remarks concerning approximations to optimal 0-order algorithms are appropriate.

The situations in which the algorithm designated $A_0$ will provide a reasonable approximation to optimality are those in which the page reference probabilities are slowly varying. Let $T_m$ denote the maximum settling time for memories of size $m$, that is,

$$T_m = \max_{|S|=m} T(S).$$

We take "slowly varying" to mean that measured estimates $\hat{p}(i, t)$ of the probabilities $p(i, t)$ define a ranking relation that is expected to hold at least through $t + T_m$. (This is consistent with the "principle of locality" [4, 5, 13], which states that programs tend to concentrate their references in particular subsets of $N$ for relatively long times.) We shall not attempt to account for the error in the estimates, $|\hat{p}(i, t) - p(i, t)|$.

Now, let $L_t^m$ denote the $m - 1$ highest ranked elements of $N$ at time $t$. Suppose $L_{t-1}^m \subset S_{t-1}^m$ (i.e., $A_0$ is in steady state at time $t - 1$), but $L_t^m \not\subset S_t^m$. If

$$h = |S_{t-1}^m - L_t^m|,$$

then Lemma 3 assures that $A_0$ will reestablish steady state with additional cost at most $h$. In other words, if the program changes the set $L^m$ by $h$ pages, $A_0$ will "recover" at cost of at most $h$. In general, if the expected number of such changes per unit time is $K$ and the expected size of the change is $H$, then

$$|C_k(A_0, S, t) - C_k(S, t)| \leq HKk.$$

Therefore, the slower the variation in the probabilities—reflected as smaller values

of $K$—the smaller the difference between the cost generated by $A_0$ and the optimal cost.

Among the replacement strategies discussed in the literature, there are two which act as approximations to $A_0$ for slowly varying probabilities, without requiring attempts to estimate them. These will be designated by LRU (least-recently-used—see Example 1) and WS (working set—only nonworking-set pages may be replaced). To verify this for LRU, let $\tau(t, x) + t$ be the expected $t' \geq t$ at which $r_{t'} = x$ next, and $t - \tau'(t, x)$ be the expected $t' < t$ at which $r_{t'} = x$ last. If the $p(x, t)$ are slowly varying, then $p(x, t) < p(y, t)$ suggests both $\tau(t, x) > \tau(t, y)$ and $\tau'(t, x) > \tau'(t, y)$. It follows that the page $s$ with the longest expected time since last reference, $\tau'(t, s)$, is the most likely to be the one with the largest value of $\tau(t, s)$. Therefore, only the pages with smaller values of $\tau'(t, i)$, i.e., larger values of $p(x, t)$, will tend to remain in memory.

To verify that WS behaves as an approximation to $A_0$ for slowly varying probabilities, define the working set at time $t$ for parameter $h > 0$ to be

$$W(t, h) = \{x \in N \mid r_k = x \text{ for some } k, \quad t - h \leq k < t\}.$$

In other words, $W(t, h)$ is the "contents" of a "window" of size $h$ looking back at the reference string. Clearly, the pages with the largest values of $p(x, t)$ will tend to be members of $W(t, h)$.

## 6. *Conclusions*

The primary objective of this paper has been to formalize certain aspects of the memory allocation problem, one of some importance in computer system design and analysis. Starting from the statement of the informal principle of optimality (replace the page with the longest expected time until reuse), we developed formal definitions for paging algorithms and "cost" and constructed a dynamic programming formulation of the minimum achievable cost. For programs that generate their reference strings by an "almost" stationary 0-order Markov process, we were able to exhibit an optimal paging algorithm and study its properties. We concluded with a discussion of algorithms that approximate optimal algorithms in slowly varying nonstationary cases.

Many questions remain unanswered. For example: How does the error in page reference probability estimates affect the quality of $A_0$'s approximation to the minimum cost? How "robust" is $A_0$—that is, if $A_0$ is applied to an $l$-order program, how well does it approximate the $l$-order cost? What optimal algorithms exist in $l$-order cases? For various $l$-order assumptions, how close to optimal are algorithms based on the informal principle of optimality? We hope to have provided a step toward answering these questions.

## REFERENCES

1. KILBURN, T., EDWARDS, D. B. G., LANIGAN, M. J., AND SUMNER, F. H. One-level storage system. *IRE Trans. EC-11*, 2 (April 1962), 223–235.
2. DENNIS, J. B. Segmentation and the design of multiprogrammed computer systems. *J. ACM 12*, 4 (Oct. 1965), 589–602.
3. ARDEN, B. W., GALLER, B. A., O'BRIEN, T. C., AND WESTERVELT, F. H. Program and address structure in a time sharing environment. *J. ACM 13*, 1 (Jan. 1966), 1–16.
4. DENNING, P. J. Virtual memory. *Comp. Surveys 2*, 3 (Sept. 1970), 153–189.

5. BELADY, L. A. A study of replacement algorithms for virtual storage computers. *IBM Sys. J. 5*, 2 (1966), 78–101.
6. FINE, G., JACKSON, C. W., AND MCISSAC, P. V. Dynamic program behavior under paging. Proc. 21st Nat. Conf. ACM, 1966, Psychonetics, Narbeth, Pa., pp. 223–228.
7. SHEMER, J. E., AND SHIPPEY, B. Statistical analysis of paged and segmented computer systems. *IEEE Trans. EC-15*, 6 (Dec. 1966), 855–863.
8. CORBATO, F. J. A Paging experiment with the multics system. Proj. MAC Rep. MAC-M-384, M.I.T., Cambridge, Mass., May 1968.
9. COFFMAN, E. G., AND VARIAN, L. C. Further experimental data on the behavior of programs in a paging environment. *Comm. ACM 11*, 7 (July 1968), 471–474.
10. SMITH, J. L. Multiprogramming under a page on demand strategy. *Comm. ACM 10*, 10 (Oct. 1967), 636–646.
11. PINKERTON, T. Program behavior and control in virtual storage computer systems. Ph.D. thesis. CONCOMP Proj. Rep. No. 4, U. of Michigan, Ann Arbor, Mich., Apr. 1968.
12. DENNING, P. J. The working set model for program behavior. *Comm. ACM 11*, 5 (May 1968), 323–333.
13. ——, CHEN, Y. C., AND SHEDLER, G. S. A model for program behavior under demand paging. Report RC-2301, IBM T. J. Watson Research Center, Yorktown Heights, N. Y. (Sept. 1968).
14. HELLERMAN, H. Complementary replacement—A meta scheduling principle. Proc. 2nd ACM Symposium on Operating System Principles, Psychonetics, Narbeth, Pa. 43–46.
15. MATTSON, R. L., GECSEI, J., SLUTZ, D. R., AND TRAIGER, I. L. Evaluation techniques for storage hierarchies. *IBM Sys. J. 9*, 2 (1970), 78–117.
16. BELADY, L. A., NELSON, R. A., AND SHEDLER, G. S. An anomaly in the space-time characteristic of certain programs running in paging machines. *Comm. ACM 12*, 6 (June 1969), 349–353.

RECEIVED JANUARY, 1970; REVISED JUNE, 1970