1-1-1983

A study of paging algorithms.

William H. Wu

Recommended Citation

Wu, William H., "A study of paging algorithms." (1983). *Theses and Dissertations.* Paper 1938.

A STUDY OF
PAGING ALGORITHMS

by

William H. Wu

A Thesis
Presented to the Graduate Committee
of Lehigh University
in Candidacy for the Degree of
Master of Science
in
Computing Science

Lehigh University
1983

ProQuest LLC.
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106 - 1346

Certificate of Approval

This thesis is accepted and approved in partial fulfillment of the requirements for the degree of Master of Science.

Sept. 15, 1983
(date)

Professor in Charge

Head of the Division

Acknowledgment

The author wishes to thank Professor Samuel L. Gulden for his helpful suggestions and thoughts in preperation of this thesis.

Table of Contents

Table of Contents

Table of Contents

Lehigh University

1983

ABSTRACT

This thesis considers computer systems with two memory levels, main memory and secondary memory. Main memory and secondary memory are both partitioned into fixed size blocks called page frames. During the execution of a program, some of these page frames will contain information and the block of information contained in a page frame is called a page. The execution of the program will require the passing of pages back and forth from main memory to secondary memory. The process whereby this is done, in order to be reasonably efficient, necessitates a careful study of what can be called paging algorithms. During the execution of a given program, a program behavior has a reference string

$$w = r_1, r_2, \ldots, r_t \ldots \quad t \geq 1$$

where r_t is the page required in main memory at time t. If the required page is not in main memory, this situation is called a page fault. A paging algorithm must deal with three policy issues: fetch policy, placement policy, and replacement policy. In dealing with these, the fetch policy is usually implemented by demand paging whereas for the other two policies there are a number of strategems which may be used. These are, least recently used, least frequently used, first in first out, last in first out, among others. With all of these we try to establish optimal

algorithms. Although in practice one cannot completely optimize the process, in general one can provide reasonably good algorithms based on a mathematical analysis of the paging process. In order to carry out this analysis, several functions are introduced.

(i) The forward distance: For any page x, $d(x)$ is the least $k \geqq o$ such that $r_{t+k} = x$. If no such k exists $d(x) =$ infinity.

(ii) The backward distance: For any page x, $b(x)$ is the least $k \geqq o$ such that $r_{t-k} = x$. $b(x) =$ infinity if no such k exists.

During the execution of a program let S_t be the memory state at time t, and let q_t be the control state at time t. On the action of a given algorithm A the operation of the system will be given by a transition function.

(iii) $$(S_{t+1}, q_{t+1}) = g_Z(S_t, q_t, r_t)$$

To examine the replacement policy we introduce a replacement function:

(iv) $R(S_t, q_t, r_t) = y$ where y is to be the page removed for the memory state S_t, control state q_t and referenced page r_t.

In terms of these functions, for example, the least recently used replacement policy may be described by

$$R(S_t, w_t, r_t) = y \text{ iff } b(y) = \max_{z \varepsilon S} [b(z)]$$

By using these functions and assumptions about them as well as various cost functions, the analysis of some paging algorithms can proceed. The analyses in this thesis follow the material in [1], [2] below.

1. Aho, A. V. Denning, P. J., and Ullman, J. D. Principles of optimal page replacement. J. ACM 18, 1 (Jan. 1971), 80-93.

2. Ingargiola, G. Korsh, J. F. Finding optimal demand paging algorithms. J. ACM 21, 1 (Jan. 1974) 40-53.

1. Introduction

The purpose of this thesis is to discuss some aspects of paging algorithms. The discussion is somewhat informal though we state some of the main facts as theorems. The theorems are not proved though we hope the discussion makes them plausible. Our discussion is based on [1], [2].

We limit our attention to a computer system with two memory levels , main memory and secondary memory. The main memory is in the machine itself and the secondary memory is in the disk or drum which is connected to the machine. The machine can operate only on information in the main memory. If the information in the secondary memory is needed then it has to be brought into the main memory.

We define the page as a certain size of information, and the page frame is a block of contiguous location addresses with a certain size in the computer main memory. The size of both page and page frame consist of c cells.

We define N as the program name space which is the space that the program occupied in the secondary memory. N is divided into n pages.The system main memory space M is the space that is authorized to the programing job in the main memory. M is divided into m page frames.

N is a set of pages indexed by 1,2,3.......,n we write

$N=\{1,2,3,\ldots\ldots,n\}$

and similarly we regard M is a set of page frames indexed 1,2,3,....,m we write

$M=\{1,2,3,\ldots\ldots,m\}$

We only care about those indices of pages or page frames in the system, we don't care about the information inside any page or page frame.

If the user's program requires n pages, but only m page frames are available where m<n, then the program can not be fitted in the main memory . Then we copy m pages of the program into the m page frames in main memory. Of course if n=m we can copy the entire program into main memory.

We define the time parameter t as a discrete parameter where t=1,2,3,..... represent the instants of processing the program . A program behavior for a given program is described in a machine independent way by its reference string

$w=r_1,r_2,\ldots\ldots r_t,\ldots\ldots\ r_t\ N\ t>=1.$

The reference string is a time sequence of pages r_t where r_t is the page which is referenced at time t. That is the page r_t is needed in by the machine at time t. If the page r_t is not in the main memory at time t then it should be brought into main memory to make it available for the machine.

At each moment of time t there is page map

$f_t : N \dashrightarrow M \cup \{0\}$

where

$f_t(x)=y$ if the page x resides in the page frame y at time t.

$f_t(x)=0$ if the page x is missing from main memory at time t.

We use the term page fault to indicate a situation in which a referenced page is not in the main memory, and the memory is full. Then we have to choose a page to be removed to make the space available for the page which is referenced. The page fault rate F(w) of a reference string w is the number of page faults encountered in processing the reference string when the length of the reference string is known.

2. Paging Algorithm

A paging algorithm is an algorithm for moving pages between secondary memory and main memory. In developing such an algorithm there are three policies:

1. Fetch policy: determine which page is to be brought into main memory, and when that will occur.
2. Placement policy: choose an available target page frame into which a fetched page is to be stored.
3. Replacement policy: choose the page or pages to be removed from main memory in order to make space available for new pages.

A paging algorithm A provides the mechanism for processing a reference string $w=r_1,r_2,\ldots\ldots,r_t,\ldots.$ and generating a sequence of memory states $S_1,S_2,\ldots,S_t,\ldots$ For a given time t, if a page fault occured, let x be the page referenced and y be the page removed then $S_{t+1}=S_t+x-y$. If no page fault occured then $S_{t+1}=S_t$. Space S_0 is the initial state.

A **demand paging algorithm** is one in which a page is fetched from secondary memory only when the required page is missing from main memory. Removal of a page occurs only when the main memory is full.

Let $g_A(S,q,x)$ be the transition function which describes the change of memory and control states under a page algorithm A. If the memory state is S, the contol state is q and the page referenced is x then after the application of A the memory state becomes S' and control state becomes q'. We write $g_A(S,q,x)=(S',q')$.

For a demand paging algorithm and given m>0 the transition function has the properties:

If $g_A(S,q,x)=(S',q')$ then

$S'=S$ if $x \in S$

$S'=S+x$ if $x \notin S$ and $ABS\{S\}<m$

$S'=S+x-y$ if $x \notin S$ and $ABS\{S\}=m$

Here ABS[S] is the number of pages in main memory at state S. We define the forward distance d(x) at time t as the number of time periods to the first occurence of x from t in the reference string.

$d(x)=k$ if r_{t+k} is the first occurence of x in $r_t, r_{t+1}, \ldots\ldots$

$d(x)=\text{infinity}$ if x never occurs in $r_t, r_{t+1}, \ldots$

We define the backward distance b(x) as the number of time periods to the last occurence of x from t in the reference string.

$b(x)=k$ if the last occurence of x was r_{t-k} in $r_1, r_2, \ldots, r_t$

b(x)=infinity if x does not occur in $r_1, r_2, \ldots, r_t$

We define the replacement function $R(S,q,x)$ as the page to be removed when memory state is S and control state is q the page referenced is x. If the page removed is y then $R(S,q,x)=y$.

Now let us introduce some further special paging algorithms.

LRU(least recent used) : The page which is replaced is the page with largest backward distance .

Thus $R(S,q,x)=y$ if and only if $b(y)=\max_{z \in S} [b(z)]$

B_0 (Belady optimal algorithm) : The page which is replaced has largest forward distance. $R(S,q,x)=y$ if and only if $d(y)=\max_{z \in S} [d(z)]$

LFU(least frequently used): The page replaced is the page having received the least number of references. Let $f_t(x)$ denote the number of references to x in $r_1, r_2, \ldots, r_t$. Then

$R(S,q,x)=y$ if and only if $b(y)=\max_{z \in S''} [b(z)]$

where $z \in S''$ if and only if $f_t(z)=\min_{u \in S} [f_t(u)]$

FIFO(first-in-first-out): The page replaced is the one which has been in memory for the longest time. Define $g_t(z)=i$ as the largest integer less than or equal to t such that $S_i - S_{i-1} = r_i = z$. Then

$$R(S,q,x)=y \text{ if and only if } g_t(y)=\operatorname{Min}_{z \in S} [g_t(z)]$$

LIFO(last-in-first-out): The page replaced has been in the main memory for the least time.

$$R(S,q,x)=y \text{ if and only if } g_t(y)=\operatorname{Max}_{z \in S} [g_t(z)]$$

3. Optimal algorithm

3.1 Cost function

The cost generated by a paging algorithm A, operating on the reference string $w=r_1,r_2,\ldots,r_T$ in the memory of size $m>0$ is denoted by $C(A,w,m)$. $C(A,w,m)$ is the total time the algorithm A takes in transferring the pages in and out of main memory of size m while processing the reference string w. Suppose h(k) is the time it takes for a single secondary memory transition involving a group of k pages. If $S_{t+1}=S_t-X_t+Y_t$ where $ABS[X_t]$ is the number of pages involved in the transition, then

$$C(A,w,m)=\sum h(ABS[X_t])$$

If A is a demand paging algorithm then $ABS[X_t]<=1$. So it is a one page or no page transition for each reference. Then

$$C(A,w,m)=\sum ABS[X_t]$$

For, the transition takes time T_w+T_t where T_w is waiting time and T_t is the transmission time between main memory and secondary memory . A k-page transition will take time $k(T_w+T_t)$ if the secondary memory uses electronic selection, that is core memory, or time $T'_w+T'_t$ if the memory uses rotational selection, that is disk or drum.

Normally $T'_w > T_w$. In the case that the page is in the core memory we would have $h(k)=k$. In the case that the page is in the disk or drum then $h(k)<k$ and $h(k)=1+a(k-1)$ where $0 \le a \le 1$

> **Theorem 1:** Suppose that $h(k) \ge k$ for a given algorithm A then there exists a demand paging algorithm A' such that
> $C(A',w,m) \le C(A,w,m)$

3.2 Optimal replacement

An optimal algorithm is an algorithm which minimizes the cost function for any reference string and at any size of memory. We wish to minimize page transfers. Hence when memory is full we want to choose as a page to be removed one which either will never be referenced again or one which will not be referenced for a relatively long period of time. We don't want to move a page in and out while the other pages stay in the main memory without being referenced. In the demand paging algorithm at a given time one page is referenced and another page is removed. So an optimal algorithm is one in which we try to find the "best" choice of page to be removed.

As mentioned above the "best" choice of a page to be removed is the one with longest expected time until its next reference, that is the one with longest expected forward distance.

4. Stack algorithm

An algorithm A is called a stack algorithm if its memory states satisfy the inclusion property:

S(m,w) included in S(m+1,w) for all m, and w.

where S(m,w) is the state of the memory of size m after the reference string w has been referenced. It is also called "stack".

The inclusion property is equivalent to the following statement which indicates in more detail the structure of S(m,w).

For each w there exists a permutation of N $s(w)=\{s_1(w),s_2(w),......,s_n(w)\}$ where s_i is the ith page after the application of w such that for all m>0 $S(m,w)=\{s_1(w),s_2(w),....,s_m(w)\}$. S(m,w) is the top most m pages of s(w) . It is clear that S(0,w)= and $\{s_i(w)\}=S(i,w)-S(i-1,w)$. The vector S(m,w) is called a stack vector or just a stack. If $i<j$ then $s_i(w)$ is said to be higher in the stack then $s_j(w)$ and $s_1(w)$ is the page on the top of the stack.

The stack distance D_x for page x is the position that the page occupies in the stack S(w). If $s_k(w)=x$ then $D_x(w)=k$ else if x is not in the stack then $D_x(w)$ is infinity.

A stack algorithm has three basic properties:

P1.. The most recently referenced page is at the top of the stack. $D_x(wx)=1$

P2.. An unreferenced page will never move higher on the stack. i.e.

$D_y(w)<=D_y(wx)$

P3.. Pages below the one referenced remain fixed on the stack.

$s_k(wx)=s_k(wx)$ if $D_x(w)<k$

Thus an algorithm is a stack algorithm if and only if $R(S+y,q,x)=R(S,q,x)$ or y if x is not in $S+y$.

5. Priority algorithm

A paging algorithm is called a priority algorithm if there exists a mapping that associates with each reference string $w=r_1,r_2,.....r_T$, a sequence of linear orderings P_1, P_2, ..., P_{T-1} such that

1. P_t $(1<=t<=T)$ is an ordering of distinct pages in $r_1,r_2,...$ $...,r_T$
2. For all $m>=1$ if r_{t+1} $S(m,r_1,r_2,...,r_t$ and $ABS[S(m,r_1 r_2 .....r_t)]=m$ $1<=t<=T$ then the page in $S(m,r_1 r_2 ...r_t)$ which is replaced is given by the least element of P_t contained in $S(m,r_1 r_2 .....r_T)$.

Now let $Min_{P_t}[S]$ denote the least element of P_t contained in S.

Then

$R(S_t,q,r_{t+1})=Min_{P_t}[S_t]$ if $ABS[S_t]=m$ and r_{t+1} not in S_t.

$R(S_t,q,r_{t+1})=0$ if $ABS[S_t]<m$ or $r_{t+1} \in S_t$

The LRU, B_0, LFU, and LIFO algorithm are priority algorithms whose priority lists order pages , respectively, by increasing backward distance, increasing forward distance, decreasing frequency of use and increasing times of entering main memory. For each different priority algorithm there is a different priority list.

A priority algorithm is a stack algorithm

for,

$R(S+y,\ q,\ x)=\mathrm{Min}[S+y]=\mathrm{Min}[\mathrm{Min}[S],\ y]$

since $R(S,q,x)=\mathrm{Min}[S]$ so

$R(S+y,\ q,\ x)=\mathrm{Min}[R(S,\ q,\ x),\ y]$

where $R(S+y,\ q,\ x)$ is either $R(S,\ q,\ x)$ or y. The converse is also true. This is the same conclusion we had in the last chapter.

Stack updating procedure

Let $S(w)$ and $S(wx)$ be two successive stacks and suppose $D_x(w)=m$. Let x be the last page of the stack that is referenced. After the reference the stack becomes as follows:

1. $s_i(wx)=x$ if $i=1$
2. $s_i(wx)=\mathrm{Max}[s_i(w),\ \mathrm{Min}[S(i-1,w)]]$ if $1<i<m$
3. $s_i(wx)=\mathrm{Min}[S(m-1,w)]$ if $i=m$
4. $s_i(wx)=s_i(w)$ if $i>m$

Line 1 is from P1 in the last chapter, i.e. the first page is the page most recently referenced. Line 4 is from P3, i.e. those pages below the page referenced stay unchanged. The line 2 indicates those pages were not removed from the stack, where Min[R(S,q,x),y] is the page to be removed and Max[R(S,q,x),y] indicates those that are not removed . In line 3 i=m is the position vacated by the page referenced and filled by the page replaced from the set of m-1 pages above it in the stack.

The algorithm stack and priority list are identical at each moment of time only if the algorithm is LRU. That is if $S(w)=[s_1(w),s_2(w),\ldots,s_n(w)]$ is an LRU stack then $i<j$ implies that $s_i(w)$ was more recently referenced than $s_j(w)$ in w.

6. Independent reference model

In this chapter we shall consider the performance of a paging algorithm in terms of an expected page fault rate. The reference string w is a sequence of independent random variables with common stationary distribution $\{b_1, b_2, \ldots, b_n\}$ such that $P_r(r_t=i)=b_i$ for $i>=1$. Let the random variable $d_t(x)$ denote the forward distance of x right after r_t has been referenced.

$$P_r[d_t(x)=k]=b_x(1-b_x)^{k-1}$$

where the mean of $d_t(x)$ is $1/b_x$

A_0 is the algorithm in which the choice of page replaced is the one in the memory whose expected forward distance is the greatest, that is the one for which b_x is least. If we let the pages be numbered so that $b_1>=b_2>=\ldots>=b_n$ the replacement rule of A_0 is

R(S, q, x)= the largest numbered page in S

We use the theory of Markov chains to analyze the LRU paging algorithm. Let $\{S_i\}$ i=1,2,3,.... be the sequence of stacks generated by the LRU algorithm for a reference string where the memory size is m. The states of a Markov chain are then the topmost m pages on the stack. The set Q consists of all the permutation of m elements taken from N. The transition probability

$P(S, S')=P_r[S_i=S \mid S_{i-1}=S]$

If $S=[j_1,j_2,\ldots,j_m]$ and $S'=[k,j_1,\ldots,j_{i-1},j_{i+1},\ldots,j_m]$ where $k=j_i$ then $P(S,S')=b_k$

Because $\{S_i\}$ is irreducible, that is, for each S and S' there exists a positive number k such that $P^k(S,S')$ is the probability of passing from state S to state S' in k transitions, and P is non=zero. This implies

$P^m(S,S')=b_{ji}>0$

for all S and S' in Q

Let (π) denoted the equilibrium probability vector and P be the transition probability matrix then $(\pi)=(\pi)P$. Let π_s and $S=[j_1,j_2,\ldots,j_m]$ denote the equilibrium probability of state S then $\{\sum(\pi)_s\}=1$. Let $P_f(S)$ denote the probability that a page fault occurs then

$F(LRU)=\sum P_f(S)(\pi)_s$ where F(LRU) is the number of page faults of the LRU algorithm.

Theorem 1: For the independent reference model

$$F(LRU)=\sum_{S\in Q} D_1^2(S)\prod_{i=1}^{m} b_{j_i}/D_i(S)$$

where $S=[j_1,j_2,\ldots j_m]$ and $D_i(S)=1-\sum_{k=1}^{m+i-1} b_{j_K}$

7. LRU stack model

Let $S(w)=\{s_1(w),s_2(w),\ldots s_n(w)\}$ where $s_i(w)$ is the i-th most recently referenced page that is S(w) order the pages according to increasing backward distance. Let $D_x(w)$ denote the position of page x in the stack s(w). We were able to associate the distance string $D_1, D_2, \ldots, D_t, \ldots$ in the reference string.

If we let D_t be the distance of r_t in the stack S_{t-1} then D_t becomes the number of distinct pages referenced since the most recently referenced page is r_t. The LRU-stack distance string is considered to be a sequence of independent random variables governed by a stationary probability mass function

$P_r[D_t=i]=a_i \quad i=1,2,\ldots,n$

whose cumulative distribution function is given by

$$A_i = \sum_{j=1}^{i} a_j$$

Let $I=i_1,i_2,\ldots,i_t,\ldots$ be the sequence of sample values for the random variables and let $S_0,S_1,\ldots S_t,\ldots$ be the corresponding LRU-stack sequence with S_0 the initial state. Then

$S_t=\{s_t(1),s_t(2),\ldots,s_t(n)\}$, the string generated by I, is defined as $w=r_1, r_2, r_3, \ldots, r_t, \ldots$ $r_t=s_t(1)$ with the initial stack understood. For a given probability mass function $\{a_i\}$ the class of strings definable in this manner will be called the class of LRU reference strings.

If the distance distribution is chosen to be biased toward short distance that is $a_1 >= a_2 >= .... >= a_n$ then the reference string will exhibit a tendency to cluster reference to the pages near the top of the stack. Conversely, if the distance distribution is biased toward long distances then , the reference string will tend to exhibit random scattering of references across many pages. The LRU paging algorithm is optimal for a class of LRU reference strings for m>=1 whenever the distance distribution satisfies

$$a_1 >= a_2 .... >= a_n$$

8. More study on optimal paging algorithm

We have defined demand paging algorithms and for those we have introduced-- FIFO, LRU, ... etc are demand paging algorithms . Now we turn our attention exclusively to demand paging algorithms. We do this for two reasons. First, with certain constraints on memory system organizations, an optimal paging policy must a demand policy. Second, a great number of systems for which a demand policy would in theory be optimal are committed by their implementation to using demand paging only.

We now introduce another non-stationary Markov process, which is called a program.

Definition 1: A program P is a system with five components. They are N, U, u_0, f, p. N is the set of pages, U is the set of program states, and u_0 is the initial state which is included in U, where f is the state transition function $f:N*U \dashrightarrow U$, p is the probability function $p(x,u,t)$ which is the probability at the time t that the page referenced is x, and the program state is currently at u . For each u U and $t>0$ then $(\Sigma p(x,u,t))=1$. The program P generates a reference string $r_1, r_2, \ldots r_T$ as follows:

For any $t>=1$, r_t has the value x with probability $p(x,u_{t-1},t)$ and $u_t=f(r_t,u_{t-1})$

The program is said to be an 1-order program if

ABS[U]=l+1 that is the program has l states besides the initial state. It is stationary if the probability function p is independent of time t.

Example:

Consider a program whose states consist of $U=N\cup\{u_0\}$ with transition function $f:N*U\text{---->}U$ given by $f(x,u)=x$. Clearly the program is an n-order program. If $p_{ij}(t)=P_r[r_t=j|r_{t-1}=i]$ then take $p(x,u,t)=p_{ut}(t)=P_r[r_t=x|r_{t-1}]$. Clearly $\sum p(x,u,t)=\sum p_{ux}(t)=P_r[r_t=x|r_{t-1}]=1$

We now take another look of the cost function in the paging algorithm which is executing the reference string. Here the reference string was generated by the program. We let $w=r_1, r_2,\dots, r_t,\dots,r_{t+k}$ $u_t=u$ and $S_t=S$. We define the cost function $C_k(S,u,t)$ for k references beyond time t recursively as follow:

$$C_k(S,u,t)=\sum p(x,u,t+1)$$

$$*\,C_{k-1}(S,f(x,u),t+1) \qquad \text{if } x\in S$$

else if x is not in S then

$$*[1+\text{Min}\ C_{k-1}((S+x-z),f(x,u),t+1)]$$

When $x \in S$ then there is no transaction and t moves to t+1, k becomes k-1. If x is not in S then there must be a transition as the result of a page fault, thus there is an increment of at least 1. The algorithm has not been completely specified. We use "Min" to designate any algorithm which can minimize the cost. We may call this the optimal algorithm.

> **Definition 2:** 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 a reference string of length T is determined by a program $P=(N,U,u_0,f,p)$ which is l-order. We denote such an algorithm by A_l and call it an l-optimal program .

A_l is much too difficult to implement since it requires both the knowledge of the probabilities as well as of the reference string. The latter may not be known in advance. However the case $l=0$ can be treated with some simplicity and we do that here by examining 0-order programs.

Recall that a 0-order program is a program which has only one state i.e. the initial state. We write p(x,t) instead of p(x,u,t). In this case the cost function simplifies to:

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

$$C_k(S,t)= \sum_{x \in N} p(x,t+1)$$

$$* \, C_{K-1}(S,t+1) \qquad \text{if } x \in S$$

$$\text{or} \quad *[1+ \operatorname{Min} C_{k-1}(S+x-z,\ t+1)] \qquad x \text{ not in } S$$

The almost stationary case

The almost stationary case is defined to be the case in which the probability distributions maintain their relative order with respect to time. That is if p(x,t)>=p(y,t) then p(x,t+t')>=p(y,t+t') for all t>=0. Under this circumstance, as given in the definition below, we can define a binary relation < on N such that we can obtain the smallest element of any S.

Definition 3: A stationary ranking relation < is a binary relation on N such that x<y if and only if p(x,t)<=p(y,t) for all t>0 . The notation x<=y means x=y or x<y where x=v means p(x,t)=p(y,t). The notation s=Min S means s S and s<=x for all x S.

Observe the following consequences of the definition.

Lemma 4: For some t>0 and S' included in N x<y implies $C_k(S'+x,t)>C_k(S'+v,t)$. Then if s=Min S then $C_k(S-s,t)=\operatorname{Min}_{z \in S}(S-z,t)$.

Lemma 5: Suppose $<$ is the stationary ranking of N and $x<y$ then $1>=C_k(S+x,t) - C_k(S+y,t)>=0$ where x,y are not in S and $t>0$.

Theorem 6: If the program P of 0-order has the stationary ranking $<$ on N with 0-optimal algorithm A_0, then the optimal algorithm has the map g_{A_0} given by

$g_{A_0}(S,x)=S$ if $x \in S$
$=S+x-s$ if x is not in S
where $s=\mathrm{Min}\ S$ and $ABS[S]=m$

8.1 Property of the optimal algorithm

Theorem 7: Suppose $<$ is a stationary ranking of N and its correspondent algorithm A_0 with $ABS[M]=m$ generates $\{S_t^m\}$ $t>0$ for some reference string w. If S_0^m is included in S_0^{m+1} then S_t^m is included in S_t^{m+1}.

S_t^m is the memory state at time t where the memory size is m. Thus the memory states S_t^m satisfy an inclusion property. This is just the same as for the stack algorithms we have discussed. Thus we can see under this circumstance that the optimal algorithm is a stack algorithm.

Definition 8: Define the set L^m to be the set of the m-1 highest ranking pages in N. The memory is said to be in a steady state if and if only L^m is included in S^m_t

The settling time T(S) is the expected time required for the algorithm to enter the steady state. If T(S)=0 then we say S^m_0 is a good starting state. Although T(S) is generally not zero, it can be shown that the cost of getting into a steady state is low. Let $S=L^m-S_0$ then the settling time $T(S_0)$ for S_0 is less than or equal to $(1/p_1)$

Theorem 9: Suppose the 0-order page reference probabilities are stationary under A_0. The expected cost per reference is

$$C'(S)=\mathrm{Lim}\ C_k(S)/k = B-1/B\left(\sum_{i=1}^{n} p_i^2\right)$$

where $N=\{1,2,3,...,n\}$ and $p_1 >= p_2 ..... >= p_n$ and

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

8.2 The non-stationary case

The binary relation that is the ranking relation only can be defined for the almost stationary case or in a non-stationary case with the following restrictions. If x<y then y must appear before x in the reference string or x

does not appear at all. Define $p(x,t)=1$ if $r_t=x$ or $p(x,t)=0$ otherwise. So $0=p(x,t)<p(y,t)<=1$ if $x <_t y$. Note that $x <_t y$ implies to $x <_{t-1} y$ if r_t is not y.

9. The cost function

Here we shall define the notion of program differently.

We define a program to be a 5-tuple $P\{N,U,(\pi),A,l\}$ where $N=\{1,2,3,....\}$ is the set of pages of the program. $U=\{u_1,u_2,....,u_k\}$ is the set of program states. where $\pi=\{(\pi)_1,(\pi)_2,..(\pi)_k\}$ where $(\pi)_i$ is the probability that u_i is the initial state, $A=\{p_{ij}(t)\ i>=1\ j<=k\ t=123...\}$ is the set of transition probabilites. Recall that $p_{i,j}$ is the probability that there will be a transition from program state i to program state j at time t, and l is a mapping from program states space U into the set of pages N.

So the reference string can be considered as the set of functions of a finite state Markov chain. We now define the set of absorption states as those states which when the program enters them it never leaves. In the state space we consider we assume that there is at least one reachable absorption state in it . In the real world we can take an absorption state as a stop mode.

9.1 Cost of a given replacement policy

We now assume that finitely many pages are referenced before we meet the absorption state. We define C_q as cost of the replacement policy q for the program P which generates the reference string. This C_q is the number of page faults encountered in executing the reference string. Let $C_u(S,t)$ be the average number of page faults incurred after time t, assuming the memory state is S and the program state is u at timt t, when the program is executed under the replacement policy g. $C_u(S,0)$ is the cost if the initial state is u and the memory state is S. Let q be the optimal policy that will minimize the cost $C_u(S,T)$ for all u U and all memory states S in N and ABS[S]=m t>0.

Let $C^*_u(S,t)$ denote the optimal cost . It must satisfy the recursive definition of the cost:

$$C^*_u(S,t)= \sum_{l(x)\in S} p_{u,x}(t+1)C^*_x(S,t+1)$$

$$+ \sum_{l(x)\text{not in } S} p_{u,x}(t+1)*$$

$$[1+\mathrm{Min}_{z\in S}\, C^*_x(S+l(x)-z,t+1)]$$

where the page referenced is l(x).

It is clear that if x is an absorption state then $C^*_x(S,t)=0$. If g is a optimal policy then the z in the equation above can be replaced by the replacement function g(u,S,x,t).

For a given policy g we can define a matrix $Q_g(t+1)$ of the size $k\binom{n-1}{m-1}*k\binom{n-1}{m-1}$. The entry of the matrix in the position (u,S), (x,S') is the cost for the program to change it's program state from u to x and memory state from S to S' when the page referenced is l(x) under the policy g. Because the page referenced l(x) must not be in S but must be in S' we only consider m-1 pages of n-1 pages.If x is an absorption state then the entry (u,S), (x,S') will be 0. Then the equation above can be written as :

$$C_g(t)=Q_g(t+1)*C_g(t+1)+b(t+1)$$

where $b(t+1)=\sum_{l(x)\in S} p_{u,x}(t+1)$ so b(t) is indepentdent of the policy g.

Now we define $E_g(t,k+1)$ as the column matrix of the cost matrix when g has been followed by k+1 references during the time from t to t+k+1. Then of course $E_g(t,0)=0$.

$$E_g(t,k+1)=Q_g(t+1)*E_g(t+1,k)+b(t+1)$$

The sequence $E_g(t,0),E_g(t,1),.....E_g(t,k+1)$ is non-decreasing and bounded above by $C_g(t)$. Moreover C_g is the limit given by

$$b(t+1)+\sum_{k=1}[\prod_{r=1}^{k} Q(t+r)]*b(t+k+1)$$

If the transition probability of program P is independent of time t then

$$C_g=Q_g C_g+b$$

and $(I-Q_g)$ must invertable. C_g is the unique solution of the above equation which is given by $(I-Q_g)^{-1}*b$. These result are summarized by the following theorem.

> **Theorem 1:** In the case of time varying transition probabilities, the cost of a policy g is given by the minimal nonnegtive solution of
>
> $C(t)=Q(t+1)*C(t+1)+b(t+1)$. This solution is $b(t+1)+g[Q_g(t+r)]*b(t+k+1)$

9.2 How to determine the optimal cost and replacement policy

The most simple way to determine the optimal policy is to enumerate the cost of all the replacement policies and find the least one. But this is not practical, so now we present a way for searching for an optimal replacement policy in the policy space.

Assume the stationary case i.e. the probability function is independent of time t. Then the cost $C_g=(I-Q_g)*b$ where b is a constant independent of the policy g. For a specified policy g and the replacement function g(u,S,x) for all u, S, and x, the page referenced is $l(x)$. Then the new memory will be $(S+l(x)-g(u,S,x))$. We see that the policy g only specifies the page that to be removed from main memory. If we change the policy from g to g' then we change $g(u,S,x)$ to be $g'(u,S,x)$. So we define another $C'_g = C_g - C_g$. Thus

$$C_g = (I-Q_{g'})^{-1} * (Q_g - Q_{g'}) * C_g$$

where $(I-Q_{g'})^{-1} = \sum_{k=0} Q_g^k >= 0$

Consequently $C_{g'} <= C_g$ if and only if $C_g >= 0$. By the equation above we have to know the value of $(Q_g - Q_{g'})$ which is matrix which has all zeroes except in the entry position (u,S), $(x, S+l(x)-g(u,S,x))$ and (u,S), $(x,S+l(x)-g'(u,S,x))$. If the differences are r and -r then only 0, r, -r appear in the matrix. Consequently C_g can be positive only if $C_u(S+l(x)-g'(u,S,x)) <= C_u(S+l(x)-g(u,S,x))$. We obtain the conclusion:

1. C_g can be positive if at least one decision changes from $g(u,S,x)$ to $g'(u,S,x)$ and we then get the result $C_u(S+l(x)-g'(u,S,x)) < C_u(S+l(x)-g(u,S,x))$
2. C_g must be greater than or equal to 0 if each decision change satisfies condition (1).

Now we consider the replacement policy of choosing the best page to be removed at any given time. Thus there is no general rule for the replacement policy as we had in the previous chapter.

We introduce the following strategy for searching the optimal policy in the policy space.

1. Pick an initial policy say g_0 and calculate the cost $C_{g_0}=(I-Q_{g_0})^{-1}*b$. Let n=0 and go to (2).
2. If $\text{Min}\,C_u(S+1(x)-z)=C_u(S+1(x)-g_n(u,S,x))$ for all $x\in U$ or 1(x) in S then this is the case to stop, otherwise we have to keep on searching for another. Define $g_{n+1}(u,S,x)=z_0$ where $\text{Min}_{z\in S}\,C_x(S+1(x)-z)=C_x(S+1(x)-z_0)$ for all u,and S, go to (3).
3. Determine the cost $C_{g_{n+1}}=(I-Q_{g_{n+1}})^{-1}*b$ if $C_{g_{n+1}}=C_{g_n}$ then stop .
4. If $C_{g_{n+1}}<C_{g_n}$ then n becomes n+1 and go to 2.

This strategy is based on the two rules we had earlier. It stops when an optimal policy is found, otherwise it iterates again. This iteration procedure can not repeat a policy since g_n is not equal to g_{n-p} for all $0<p<=n$. Since there are only finite number of policies in the policy space the iteration must converge. This represents an efficient technique in searching through the policy space to get an optimal policy. But the choice of the initial policy may influence the speed of convergence. Of course, the choice of an optimal policy as the initial policy would be the best. If the case is non-stationary then the time-varying transition probability will make the process more complex.

9.3 Extension to a larger program

The procedure developed above was for the stationary case in a program with only few states. Now we present another method for extending to a larger program.

Let $G=\{G_1,G_2,....G_L\}$ be a partition of the program states space U of program P. Those G_i are non-empty and disjoint the union of G_i is U . The set of states reachable from G_i and not in G_i will be denoted by G'_i . Then we define the program P_i to be the program whose set of pages is $N_i=\{l(x)|x \text{ in } G_i \text{ or } G'_i\}$, and the program state set is the union of G_i and G'_i, the transition probabilities are are those of A restricted to G_i and denoted by A_i . The states G'_i are absorption states .

Let $C_L(l)$ be the column matrix of the optimal cost of the program P_L under an optimal policy. If u is in the intersection of G'_{L-1} and U_L then it is an absorption state of U_{L-1}. Then the cost of u can be assigned from the appropriate component of $C_L(1)$. Hence all of the absorption states in U_{L-1} will be assigned a cost not necessarily zero. C_{L-1} denotes the column matrix of the cost for the program P_{L-1} under an optimal policy.Then this procedure is carried out for P_{L-1}, P_{L-2}, ... , P_1 each time on the sets G'_i of external states G'_i of G_i. Thus each $P_1, P_2, \ldots P_L$ will be be considered once, with a certain cost and policy determined for it.

Let's consider the program itself. For $u \in G_i$, g any policy, then the cost function can be written as:

$$C_u = \sum_{x \in G_i \quad l(x) \in S} p_{u,x} C_x(S)$$

$$+ \sum_{x \in G_i \quad l(x) \text{ not in } S} p_{u,x} [1+C_x(S+l(x)-g(u,S,x))]$$

$$+ \sum_{x \in G'_i \quad l(x) \in S} p_{u,x} C_x(S)$$

$$+ \sum_{x \in G'_i \quad l(x) \text{ not in } S} p_{u,x} [1+C_x(S+l(x)-g(u,S,x))]$$

Q_g is the matrix corresponding to g n $C=Q_g*C+b$. Then we define Q'_i and Q''_i by

$$(Q')_{i,(u,S),(u',S')} = (Q)_{(u,S),(u',S')} \quad \text{if } u \in G_i \ \ u' \in G_i$$

$$= 0 \text{ otherwisw}$$

$$(Q'')_{i,(u,S),(u',S')} = (Q)_{(u,S),(u',S')} \quad \text{if } u \in G_i \ \ u' \in G'_i$$

$$= 0 \text{ otherwise}$$

Then $Q_g = \sum_{i=1}^{L} Q'_i + Q''_i$, Q'_i and Q''_i are 0 unless they correspond to G_i and G'_i. Then

$$C_u(S) = \sum_{i=1}^{L} (Q'_i + Q''_i)*C+b$$

If Q represents the optimal policy for P then $Q'_i+Q''_i$ would represent the optimal policy for P_i with the external states G'_i of G_i. We let C_u^{n+1} be the cost for the policy g_{n+1} then it is the (u,S)-th component in the C(n+1) which is the cost of P_i under replacement policy g_{n+1}. After the n'th iteration of the approximation procedure , the sets G'_i of external states G'_i of G_i was assigned the non-zero value cost. Let Q(n+1) be the matrix for the policy g_{n+1} then

$C(n+1)=Q(n+1)*C(n+1)+b(n+1)$ where $b(n+1)=b+D(n+1)$. $D(n+1)$ represents the contribution to $C(n+1)$ of the component of $C(n)$ corresponding to the set of external states G'_i of G_i $i=1,2,3,.....L$. Define $Q'_i(n+1)$ and $Q''_i(n+1)$ in a manner similar to Q'_i and Q''_i respectively then

$$C(n+1)=Q'(n+1)*C(n+1)+b+D(n+1)$$

where $D(n+1)=Q''(n+1)*C(n)$ $D(0)=0$.

If $Q(n+1)$ represents the optimal policy for P_i $i=1,2,3,...,L$ then $Q'(n+1)+Q''(n+1)$ would represent the optimal policy for P_i with the the set of external states G'_i of G_i assigned non-zero absorption state costs. Then

$$C_x^{n+1}(S+1(x)-g_{n+1}(u,S,x)=\mathrm{Min}_{z\in S}\ C_x^{n}(S+1(x)-z)$$

Here $b(1)=b+b(0)$ hence $c(1)>=C(0)$ $C(n+1)=(I-Q'(n+1))^{-1}*b(n+1)$, $C(n)=(I-Q'(n))^{-1}*b(n)$. Assume $b(n)>=b(n-1)$ and $C(n)>=C(n-1)$. Then $b(n)=b+D(n)$ $b(n+1)=b+D(n+1)$. By definition $D(n+1)>=D(n)$ if $C(n+1)>=C(n)$. Hence $b(n+1)>=b(n)$ and $C(n+1)>=C(n)$ for all n by induction. Since $C(n)$ and $b(n)$ are bounded, they must converge as n increass to say $\hat{C}$ and $\hat{b}$ and D is also bounded and converge to say $\hat{D}$. Then

$$\hat{b}=b+\hat{D}$$

Let $\hat{Q}'$ minimize $(I-Q^*)^{-1}*b$ over all the policies Q^*

which have the same form as $\sum_{i=1}^{L} Q'_i$. If n is large enough then $(I-Q')^{-1}$ *b(n+1) will be very close to $(I-Q'(n+1))^{-1}$ *b(n+1). We take the limit of C(n+1) as n tends to infinity then

$\hat{C}=\hat{Q}'*\hat{C}+\hat{b}$ where $\hat{b}=b+\hat{D}$

Now $\hat{D}=\hat{Q}''*\hat{C}$ for some $\hat{Q}''$ hence $\hat{C}=\hat{Q}'*\hat{C}+b+\hat{Q}''*\hat{C}$. Thus $\hat{C}=(\hat{Q}'+\hat{Q}'')*\hat{C}+b$

Two remarks should be made. First, if the partition of G has the property that for each i only those G_j can be reached from G_i where j>=i, then exactly one iteration is required in operating the approximation procedure to determine the optimal policy and its cost for the program P. Second, if for each G_i with only one state from U, then the C(n+1) for P_i only has one term, so it simplifies in solving the cost equation to get Q(n+1). However, the larger G_i of G is the faster the convergence of the iteration procedure. So we face the two options namely the cmplexity of the computation and the number of iterations. We considered these two effects and tried to reduce the total time consumed in getting the optimal cost and policy.

References:

1. Aho, A. V. Denning, P. J., and Ullman, J. D. Principles of optimal page replacement. J. ACM 18, 1 (Jan, 1971), 80-93.

2.Ingargiola, G. Korsh, J. F. Finding optimal demand paging algorithms. J. ACM 21, 1 (Jan 1974) 40-53.

VITA

The Author's name is William Wu. He was born Ho-Sun Wu in Taiwan, Republic of China, in 1952. His father's name is Tzu-Eseng Wu; his mother's name was Chung-Shu.

Mr. Wu was graduated from the high school of the National Normal University in Taiwan in 1970, and was graduated from Soochew University, Department of Business Mathematics, in 1975, with a Bachelor of Art degree.