L. A. Belady
F. P. Palermo

On-line Measurement of Paging Behavior by the Multivalued MIN Algorithm

Abstract: An algorithm is presented that extracts the sequence of minimum memory capacities (MMCs) from the sequence of page references generated by a program as it is executed in a demand paging environment. The new algorithm combines the advantages of existing approaches in that the MMC's are produced in a single pass, as is the output of the MIN algorithm for a single memory size, and the MMC sequence is identical to the optimum stack distances provided by the OPT algorithm, which requires two passes.

A hardware implementation is outlined as an extension to existing page management mechanisms. The resulting device could be used to produce continuously the MMC information, while the (paging) machine executes the program at essentially full speed. The paper also discusses the possible impact of the algorithm on the study of program behavior and on the development of space sharing (paging) algorithms. Finally, a proof is provided that the algorithm in fact produces an output identical to that of OPT.

1. Introduction

This paper describes an algorithm that computes the minimum paging overhead from the page reference string of a program without look-ahead. Called the multivalued MIN algorithm, it is equivalent to a one-pass version of OPT [1]. The immediate implication is the significantly cheaper data gathering it affords. In addition, the novel view of reference strings furnished by the algorithm suggests fresh approaches to the development and evaluation of memory management schemes, i.e., multiprogramming in a paging environment. The study of alternative approaches to multiprogramming presents more severe requirements for fast measurement tools than that of uniprogramming, due to the additional degree of freedom introduced by dynamically varying memory space.

Extensive work has been done on the analysis of memory hierarchies [2]. The reader should recall that all page replacement algorithms face uncertainty when choosing a page for removal from main memory. It was early recognized [3] that, given a program characterized by its page reference string and the size of main memory, it is useful to know the minimum number of page faults necessary to run the program in order to evaluate memory configurations and page management schemes. For example, the efficiency of a page replacement algorithm is defined as the minimum number of page faults divided by the number of page faults generated when using the particular algorithm. Efficiencies have since been extensively measured and found to vary between about 0.15 and 1.0, an approximate mean being about 0.4.

Two distinct, although necessarily related, methods have been developed to extract the minimum number of page faults. One, called MIN [3], can process the reference string as it is generated by the program and, given a fixed memory size, compute the associated single minimum page fault count by constructing, but only after a necessary and variable delay, the memory states and their transitions.

The disadvantage of MIN, that it works for a single memory size at a time, was alleviated by the other approach, the two-pass OPT stack algorithm [1]. The OPT algorithm computes the minimum page fault counts for the entire range of memory sizes essentially concurrently. The algorithm would require repetitious look-ahead except that this is eliminated by a preprocessing pass. During this first pass, performed in reverse order, the forward distance string (reflecting the order of next occurence of each element in the reference string) is constructed. The reference string is subsequently processed by OPT to compute the optimum distance string—the sequence of minimum memory capacities—by using the output of the first pass to construct a priority list.

On balance, both schemes are useful. The MIN algorithm is simple, requires no look-ahead, and is efficient when only a single memory size is of interest. It is often used in compilers for register allocation [4, 5]. The OPT algorithm, on the other hand, is more elaborate and requires a larger amount of recorded information, but it yields directly the complete space-time behavior of the program. As a result, OPT is extensively used to evaluate storage hierarchies [6].

Neither algorithm, however, has been used on-line in an operational environment. Their impact on page management schemes is therefore only indirect, through increased insight into system behavior. They offer no possibility of real-time detection and exploitation of page referencing patterns.

For the sake of completeness we mention two additional relevant pieces of work. A paper on index register allocation [7] offers a neat solution to an even more general problem than the one under study here, while a patent [8] describes an on-line device that calculates approximate values of the minimum page fault count.

The multivalued MIN algorithm introduced below requires that only a small amount of information be recorded or stored while processing, with a resulting capability of being used on-line. The elimination of look-ahead makes a hardware implementation feasible. In addition, the algorithm has already permitted the discovery of interesting program properties, some of which are suggested in section 4.

In this paper, we first describe the multivalued MIN algorithm, revealing at the same time some interesting correspondences among MIN, OPT, and the new algorithm. We then discuss a hardware implementation of the new algorithm. Some of the possibilities for detecting and using program properties, for example in multiprogramming environments, are then considered. A proof that the outputs of OPT and the multivalued MIN algorithm are identical is provided in an appendix.

2. Multivalued MIN algorithm

Consider a program represented by its page reference string R. For any replacement algorithm, such a string is the input. For a given memory size, MIN produces as output a single value, the least number of page faults required to run the program. The OPT algorithm also uses the input string R, first to produce the forward distance sequence, then to use this new sequence for the process that actually extracts the sequence of optimum stack distances. Each element of string R—the original input—becomes associated with an OPT stack distance.

Since an optimum stack distance is the minimum memory size associated with a reference such that no page fault occurs, the minimum number of faults for any fixed memory size is computable from the optimum stack distance string, as described in [1]. In this paper we do not have an explicit stack in the sense of [1] and therefore call such an output string the sequence of minimum memory capacities (MMCs). A MMC value of p then means that, prior to being referenced, the element associated with the output has been contained in memories of size p or greater and therefore must be pulled by being referenced into all memories smaller than p.

Historically the ultimate purpose of optimum stack construction in evaluating memory hierarchies has been to produce the MMC string. (The "hit ratios," i.e., the normalized page fault counts as a function of memory size, are directly computable from the MMC values; hence recording of the voluminous MMC string can be avoided.). The underlying thesis of the work presented here is that, at any point in R, the *MMC value* (or, equivalently, the OPT stack distance) *associated with the current reference is uniquely a function of the previous references; hence look-ahead is unnecessary.*

This assertion does not mean, of course, that a viable fault-minimizing replacement algorithm is possible. The knowledge of the MMC value, paired with a memory reference, is *a posteriori* information available when it is, in general, too late to assure that the referenced page will indeed be kept in a memory of this critical size. For example, if a reference to page α becomes, by our algorithm, associated with the MMC value 4, this merely means that a minimizing scheme applied from the beginning of a program running in a four-page memory would have kept α among its current four pages. In other words, 4 is the output associated with α; another page, not α, might have a different associated MMC value.

In the remainder of this section we develop the new algorithm in three phases. In the first, operation of the original MIN algorithm [3] for a given memory size is demonstrated by a two-dimensional matrix. In the next we develop the multivalued version, which is represented by a single numeral matrix that is a result of merging matrices for distinct memory sizes. This algorithm in fact produces the MMC values in one pass. The last phase then shows that a single stack is sufficient to represent the essential information contained in the numeral matrix. Thus the algorithm becomes a set of manipulations on this stack rather than on the numeral matrix of the second phase.

• *MIN algorithm*

The conceptual framework of MIN [3] is a two-dimensional matrix in which a row is associated with each page. A column represents a distinct memory state, namely, a particular collection of pages. The number of pages in such a collection may not exceed p, the memory size measured in pages. As the matrix is being constructed, a change of state occurs when a page not in

Figure 1 Successive memory states for seven references.

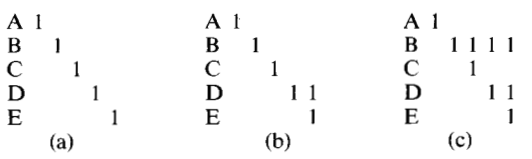


Figure 2 Memory states for a three-page memory.

	t							
A	1						1	1
B		1	1	1	1	1		
C			1			1	1	1
D				1	1	1		
E					1			1

any previously constructed collection is referenced. Re-referencing a page α can also cause a change of state if any collections have been completed with p references to other pages since the earlier reference to page α. The MIN algorithm is then a procedure for constructing a matrix representing the minimum number of states necessary to execute a program.

Consider, for example, a program having an input string R of the 14 elements ABCDEDBCBDAEAC and a memory in which $p = 3$. After processing the first five references, the information thus far available permits construction of the matrix in Fig. 1(a). Thus all five references, by definition, created a new memory state. However, we do not yet know the entire collection of pages for each state; only one element is known in each. Neither have we determined which pages will have to be pushed out of our three-page memory, the inherent delay of MIN alluded to earlier.

The sixth reference, to page D, may be added to an already defined state, namely, the fifth. Thus D and E both become members of the same collection, and only five states are required to accommodate the six references, as shown in Fig. 1(b).

The seventh reference, to page B, adds the third and final element to the fifth state, as shown in Fig. 1(c). The matrix thus represents the minimum number of page pulls (five) needed for these seven page references. It is obvious in retrospect that pages B and D should not be pushed from a three-page memory encountering this reference string. The not necessarily unique sequence of states in our example could be represented as follows: (A, 0, 0), (B, 0, 0,), (B, C, 0,), (B, D, 0), (B, D, E), where zero means empty.

A general set of construction rules (except for the first reference) can now be specified for such a matrix, given memory size p and reference string R, as follows:

- Suppose that the next reference in R is to page α and that the rightmost nonempty column is labeled $t - 1$.
- If row α is empty, mark a 1 at (α, t) and return.
- Else find the rightmost column t_α having a mark in row α.
- If there exists a column τ, $t_\alpha < \tau < t$, with p markings, mark a 1 in (α, t) and return.
- Else mark a 1 in all empty (α, τ), $t_\alpha < \tau < t$, and return.

Figure 2 shows the matrix resulting from application of these rules to the 14-element string used in our first example. (Repetitious references in a string, for example BB, would not cause new markings for any value of p and are thus redundant. For the rest of the paper, we suppress such repetitions.)

After having processed input string R in accordance with these rules, the number of marked columns is equal to the minimum number of page pulls necessary to run the program (eight in our example) for the given memory size. Unmarked gaps to the left of a marked entry (and to the left of column one for the first entry) also represent page pulls. Pages A, C, and E were pulled twice in the example. Marks in a column identify pages coexisting in memory, i.e., belonging to the same collection. The matrix produced by MIN is such that a column may contain fewer, but never more, than p pages.

Examination of Fig. 2 shows that, within the constraint set by p, gaps can sometimes be filled with markings when a new reference is made to a page previously referred to, thus avoiding a page pull. However, p markings in a column present an obstacle to a filling attempt; the gap is permanent. Recall also that columns generally acquire p markings only after a delay.

With this representation, however, it is difficult to compare and then interpret matrices generated for different p values, since the columns are generally not aligned with the input string. For the above example and $p = 5$, only five columns would be generated, the last one with all positions marked; the matrix associated with $p = 1$ would have as many columns as there are nonrepetitious page references, with each column having only one marking.

We therefore slightly modify the construction rules. In the new version of the algorithm, when the next reference occurs, we mark (α, t) for all cases. Since prior to this new reference, (α, t) has been empty, matrices for all memory sizes now become aligned with the input string, and t can be considered to be the index of time to a given

reference; the next reference is x_t and the tth column is associated with it. Thus the only addition to the former rules is to also mark a 1 in (α, t) in the last step.

All five matrices in our example, constructed using the modified rules, are shown in Fig. 3. As before, the number of markings in any column never exceeds p, but can be less. The minimum number of pulls necessary to run the program is then equal to the number of markings *not* preceded on the left by another marking, but preceded by a gap or extending to the left of the first column.

• *Multivalued MIN matrix*

We now superimpose the five matrices on one another. However, we cannot now use a single number for marking—the numeral 1—since we would have an inconclusive figure, identical to that of $p = 5$, which is the union of all five matrices. We introduce as symbols for marking the numerals 2, 3, 4, . . . , to represent the order of natural numbers.

In addition to the rules already presented, we now need some rules to guide in the use of the numerals so as to preserve information about all cases even after superimposition. To illustrate the construction rules, we proceed as follows.

First we decide to continue use of the numeral 1 in the matrix for $p = 1$; the procedure for this case is then unchanged, and there will be a 1 in as many columns as there are references processed. No other numeral is used.

Next we decide that no column may contain duplicate numerals. By this means we are able to distinguish among cases of different p values after superimposition. If a given position in the matrix for memory size p contains a mark, but the comparable position in the matrix for $p - 1$ does not, then a numeral not yet used in the relevant column of the matrix for $p - 1$ distinguishes between the two cases.

We can construct the combined matrix by sequentially processing the reference string. When the next reference x_t is to page α, we use 1 to mark (α, t). Then we note the gap extending to the left of the new marking beyond column one of the matrix (new reference) or bounded by a numeral 1 (the most recent previous reference to α). In the original version of MIN, we would have filled the gap only if, for each column in the region of the gap, the number of markings was less than some fixed p. But we are constructing a matrix for all possible p. Therefore we use the numeral for marking that indicates, in position $(\alpha, t - 1)$, the minimum p value for which the gap-filling is valid.

Since we want to find for all memory sizes the minimum number of page exceptions necessary to run the program, we must use the numerals in a specific order. Starting with the leftmost position of the gap, the smallest numeral not yet present in that column is used for marking. Since we also attach a meaning to this numeral, namely, that it designate the p-size memory capable of containing the page, a numeral once used in a gap forms a lower bound to the right in that gap, i.e., before the numeral 1 is reached. In other words, numeral string 223444 is valid while 223443 is not. The reason for the nondecreasing string is that the matrix is designed to reflect the behavior of a demand paging system. Each numeral in the string represents the minimum size memory required to retain a given page at a given time. For example, if the string above were in row α, the numeral 4 in the string would mean that a memory of at least size $p = 4$ would be required to retain page α. Page α could not be retained in the smaller memory of size 3. The appearance of a 3 in the string thus means that page α must have been pulled back into memory. But this occurs only if page α has been re-referenced, an event that in our matrix is represented by a 1.

Figure 3 Matrices for five memory sizes.

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
A	1										1		1	
B		1					1		1					
C			1					1						1
D				1		1				1				
E					1							1		

($p = 1$)

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
A	1										1	1	1	
B		1					1	1	1					
C			1					1						1
D				1	1	1				1				
E					1							1		

($p = 2$)

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
A	1										1	1	1	
B		1	1	1	1	1	1	1	1					
C			1					1	1	1	1	1	1	1
D				1	1	1	1	1	1	1				
E					1							1		

($p = 3$)

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
A	1										1	1	1	
B		1	1	1	1	1	1	1	1					
C			1	1	1	1	1	1	1	1	1	1	1	1
D				1	1	1	1	1	1	1				
E					1	1	1	1	1	1	1	1		

($p = 4$)

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
A	1	1	1	1	1	1	1	1	1	1	1	1	1	
B		1	1	1	1	1	1	1	1					
C			1	1	1	1	1	1	1	1	1	1	1	1
D				1	1	1	1	1	1	1				
E					1	1	1	1	1	1	1	1		

($p = 5$)

Figure 4 Multivalued MIN numeral matrix.

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
								t						
A	1	2	3	4	5	5	5	5	5	5	1	2	1	
B	2	1	2	2	3	3	1	2	1					
C	3	3	1	3	4	4	4	1	2	2	2	3	3	1
D	4	4	4	1	2	1	2	3	3	1				
E	5	5	5	5	1	2	3	4	4	4	4	1		
		2	3	4	5	2	3	4	2	3	5	4	2	3

Figure 5 LRU numeral matrix.

	A	B	C	D	E	D	B	C	B	D	A	E	A	C
								t						
A	1	2	3	4	5	5	5	5	5	5	1	2	1	
B	2	1	2	3	4	4	1	2	1					
C	3	3	1	2	3	3	4	1	2	3	4	5	5	1
D	4	4	4	1	2	1	2	3	3	1				
E	5	5	5	5	1	2	3	4	4	4	5	1		
		2	3	4	5	2	4	4	2	3	5	5	2	5

In the following text we call a gap-filling string starting with numeral a and terminating with numeral b an a/b string. The valid string in the example above is then a 2/4 string. Also, we exclude 1's from the strings: hence a is never 1. We now summarize the above requirements into a numeral string algorithm:

- Suppose that the next reference x_t in R is to page α. Let t_α be the time at which α was last referenced ($t_\alpha = 0$ if this is the first reference to α).
- Mark in row α every position (α, τ), $t_\alpha < \tau < t$, with a numeral q, which is the minimum missing numeral in column τ that is equal to or greater than the numeral in position $(\alpha, \tau - 1)$.
- Mark (α, t) with a numeral 1 and return.

(Note that q is redetermined for each column and that the sequence is nondecreasing.)

The application of this set of rules to our former example would produce the numeral matrix in Fig. 4. If an additional reference were now made to page B, a 3 would be inserted into column 10 of row B, since 3 is the smallest numeral not yet used in column 10 but is clearly larger than the 1 in column 9. A 3 is also used in column 11 for the same reasons. However, 3's already appear in columns 12 and 13, so a 4 is inserted into each of these. A 4 is also inserted into column 14, since this entry may not be smaller than the entry in column 13 of row B. Finally, a 1 is inserted into column 15.

The algorithm is easy to perform manually. It is equally easy to extract the uniformly marked matrix for any given value of p. This can be done in two steps:

1. Remove all a/b strings (between 1's) for which $b > p$.
2. Change all remaining symbols to 1's.

Of course, all 1's originally in the matrix are retained.

The following properties of the combined matrix are evident. If x_t is a reference to page α, then (α, t) is marked with a 1. Also, the numeral at $(\alpha, t - 1)$, say p, stands for the minimum memory capacity containing page α prior to x_t. This is the sought-after information, and the sequence of these MMC values is displayed immediately below the numeral matrix in Fig. 4. There is one MMC for every reference except for the first one; this is by definition 1, but since we eliminated repetitious references, it is suppressed.

This new version of MIN thus eliminates the look-ahead (first pass) required by OPT, yet *it produces exactly the information generated by OPT*. It is easy to see that numerals stand for (OPT) stack distances. Thus, the sequence of numerals in a given row is the history of pushes from memories of distinct sizes; a change from a to b means that the page is pushed from memories of size c, $a \leq c < b$.

It is interesting to note that a similar matrix can be made to represent the memory requirements of a program executed in a demand paging environment under a least recently used (LRU) page replacement policy. The rules for constructing the matrix are provided here for the sake of completeness; however, knowledge of this algorithm is not necessary to understand the remainder of the paper.

- Suppose that the next reference x_t in R is to page α. Let t_α be the time at which α was last referenced ($t_\alpha = 0$ if this is the first reference to α).
- Mark in row α every position (α, τ), $t_\alpha < \tau < t$, with a numeral r such that
  - * If row β, containing a 1 in column τ, contains another 1 in column λ, $t_\alpha < \lambda < \tau$, then r is the minimum missing numeral equal to or greater than that at $(\alpha, \tau - 1)$,
  - * Else r is equal to 1 plus the value at $(\alpha, \tau - 1)$.
- Mark (α, t) with a numeral 1 and return.

(Note that if x_t is the first reference in R, after marking for this reference, the matrix will contain a 1 at (α, t).)

Application of these rules to our sample page address string is shown in Fig. 5. Comparison with Fig. 4 shows the added page requirements for the LRU replacement scheme over the minimum determined by the multivalued MIN algorithm; the changed requirements are shown in boxes in Fig. 5.

The numerals in the matrix are the LRU stack positions. As rows are filled in accordance with the rules, each successive column indicates that another page has been referenced. Thus, as each column is reached, the number of pages needed to contain the currently referenced page α is usually increased by 1. However, two 1's in row β since time t_α indicate two references to the same page; hence, an additional page is not needed to retain α in the memory.

Figure 6 Minimum strings.

A					2
E				2	2
D		3	4	4	4
B	3	3	4	4	4

Figure 7 IMS and CMS.

IMS					*CMS*				
A				2	3		4		5
E				2 2			4 4		5 5
D			3 4	4 4					5 5 5 5
B		3 3	4	4 4					

• *P stack*

Unfortunately the manual ease of constructing the multivalued MIN matrix does not compensate for its bulkiness; one of its dimensions is bounded only by the length of R. Fast storage of this size is rarely available for automatic computation. We must therefore develop a more compact representation, one that keeps just that information necessary to generate the MMC sequence in response to input string R, suppressing historical information rarely used in practice (as experience thus far with OPT has indicated).

In the numeral string algorithm, upon presentation of the next reference, a new string is constructed, subject to constraints imposed by strings drawn earlier. In order to be compact, we must summarize these constraints while ensuring that needed information is retained. In particular, the rightmost numeral in the string, identifying the output associated with the page reference, must be kept (recall that 1's are not part of the string).

A simple way of doing this is to construct independently for each gap the minimum a/b string that would be drawn if these pages were referenced individually, using the rules governing the construction of Fig. 4. In the example presented there, a next reference to either B or D would produce 4 as output, whereas a reference to E or A would result in a 2 (a reference to C would be repetitious). The corresponding minimum strings, listed in the order of their lengths, or equivalently in LRU order of past references, are shown in Fig. 6.

Of course, only one of the strings could become an *actual minimum string*, the one for the page that in fact is referenced next. After that a new set of potential outputs can be defined.

Our objective is to describe the potential outputs in such a fashion that, if and when the next reference is known, not only is its associated output defined, but the description of the new potential outputs is also available.

In order to arrive at such a scheme, we note that the minimum string for D in our example is a proper right-adjusted substring of that for B; a similar relation exists between A and E. Next we observe that for strings in the longest gap there is only one starting numeral to choose from. In the next longest gap there are two free numerals, then three, and so on. In our example, 3 is the only choice of a starting numeral for a string for B, but 3 and 5 are both possible for D without duplicating numerals in a column. Similarly, 2, 4, and 5 are possible for E and finally all but 1 for A. Of course, independently constructed minimum strings drawn for individual gaps always start with the smallest numeral possible, as shown in Fig. 6.

We now define two kinds of minimum strings: the immediate minimum string (IMS) and the conditional minimum string (CMS). The *immediate minimum string* of element α is the numeral string that is generated by the rules used for Fig. 4 if the next reference is to element α. The strings for A, E, D, and B in Fig. 6 are then by definition IMSs.

A *conditional minimum string* on the other hand is a minimum string whose starting numeral is *not* the smallest possible one. A CMS is a string that is either a proper right-adjusted substring of an IMS or CMS that would have been used to fill a longer gap, or it is the minimum string that would be constructed assuming that all longer gaps have been filled. (We continue to refer to a CMS as "minimum" because it is either a substring of a longer string that is itself a minimum string or it is a string that would be minimum if all longer strings had been filled.)

The IMSs and CMSs for our example are shown in Fig. 7. The IMSs are obviously identical to those presented in Fig. 6. However, 5555 as a CMS for D, for example, is constructed under the assumption that the only possible string for B has already been drawn. Other CMSs were similarly generated. Note that each string

Figure 8 Primitive strings.

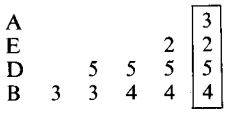


Figure 9 References to elements having primitive strings.

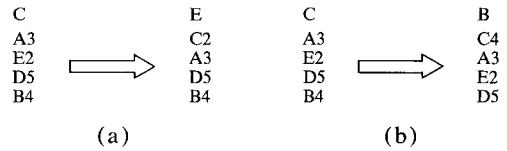
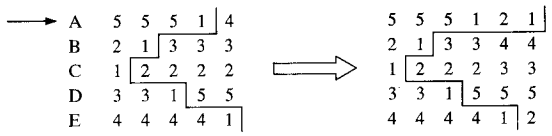


Figure 10 Effect of substrings.



(IMS or CMS) that is not a substring in the above list is underlined; we call such a string a *primitive*. Every string that is not underlined is then a substring of some primitive.

Because of this string-substring inclusion property of the IMS and CMS and the construction rules for Fig. 4, there are as many primitives as there are gaps. If n is the number of pages referenced thus far, then there are $n-1$ primitives. For our example four primitives (underlined in the figure) are constructed by application of the following rules:

- Draw the minimum string for the longest gap.
- Draw the minimum string for the next longest gap, assuming that all longer gaps have been filled with IMSs and CMSs.
- Repeat the previous step until all gaps have been filled.

By definition, the terminating numerals of the $(n-1)$ primitives contain all immediate and conditional minimum output values. The primitives for the example are shown in Fig. 8 with the terminal numerals enclosed in a rectangle. We define this LRU-ordered last column formed from the primitives as the *P stack* of output values, and we interpret it as follows.

The least recently used element B has only one starting numeral for a minimum string, and its terminal numeral is 4, which would be the next MMC output if B were referenced next.

There are two possible starting numerals for strings for D, and the corresponding IMS and CMS terminate in 5 and 4, which are listed at and below D in the *P* stack. The smaller of the two, 4, would be the MMC value associated with D if it were referenced next. Similarly, E has three possible IMS and CMS terminating in 2, 5, and 4. Therefore the smallest, 2, would be the output if E were referenced next. The MMC for A would also be 2, as the smallest of the terminal numerals 3, 2, 4, and 5.

In general, given the *P* stack of output values in the LRU order of their associated elements, the MMC value for the next referenced element can be computed, for example, as follows:

- If the next reference is to the ℓth most recently referenced element, select as MMC the smallest from the set S of numerals at or below the $(\ell-1)$st position in the *P* stack. (Note that the most recently referenced element is not represented in the *P* stack.).

In our example, if E, the third most recently referenced element, were referenced next, then the output would be 2, which is the smallest number at or below the second stack position. Similarly, the output for D would be 4, etc.

We now must show how to manipulate the *P* stack in order to reflect in it the new IMS-CMS configuration after the next reference is included. We can expect, for example, that changes may result because some CMSs become IMSs.

There are two cases to consider. In the first case the next reference is to an element whose IMS is either primitive or is a proper substring of an IMS associated with an element referenced earlier. By definition, the longest gap (that of the least recently referenced element) has an IMS that is primitive. In our example, the IMSs for B and E are primitives, while the IMSs for D and A are substrings.

Suppose that α is referenced next and that its IMS is primitive. The IMSs and CMSs for each element referenced earlier than α (below α in the LRU order) are unchanged. The reason is that, by definition of the primitive string, none of the longer IMSs or CMSs includes the string actually generated for α. The IMSs or CMSs of elements referenced more recently than α, however, are affected; either an IMS or a CMS is lost for each element in this range, because of the actual string generated in row α. But this is equivalent to moving the new output value, i.e., the terminal numeral of the actual string, from its present position to the top of the *P* stack.

If in our example E were referenced next, where the IMS for E is primitive, the corresponding stack transition would be as shown in Fig. 9(a) (where for the sake

of clarity we also list the names of the elements A through E in LRU order). Similarly, a reference to B (also with a primitive IMS) would cause the transition shown in Fig. 9(b).

Now we consider the second case. Suppose that α is referenced next and its IMS is a substring. Clearly, the IMSs and CMSs of some elements less recently referenced than α are affected, since there exists either an IMS or a CMS properly including the actual string for α. The unused portion of this including string will have to be extended to the right by new numerals. Under the minimum constraint these numerals for constructing strings are the smallest (former) CMS of α. But this CMS of α, in turn, may be a substring of some other IMS or CMS, requiring that its including string be extended by the next smallest CMS of α, etc. These effects are illustrated in Fig. 10, where the jagged lines separate the actual strings from IMSs and CMSs.

The last column in the right matrix reflects the new output (terminal numeral) configuration of all IMSs and CMSs. Putting the numerals from both matrices in LRU order results in the corresponding stack transition shown in Fig. 11.

The new output is 2—the terminal numeral of the actual string for A. Upon transition, this 2 is put on the top of the stack, indicating that, at least for now, no IMS or CMS terminates in 2 except for one IMS for the topmost element E (now the second most recently referenced element). Since C lost its IMS of 2222, its new IMS terminates in 3, which was A's smallest CMS. But this latter string is not primitive; therefore B loses its CMS of 333, which gets replaced by the 3344 string ending in 4, which was the next smallest CMS of A. Only the 55 string remains untouched.

We described earlier how to find the output MMC value in set S of the P stack when the next reference is presented. We have now seen that the updating for the new conditions requires the removal of the output value from the stack and its subsequent placement on the top. The vacated position then is filled with the next smallest numeral, if any, in set S of the original stack *above* the vacant position. This, in turn, creates a new vacant position to be filled with the next smallest in that range, if any, and so on. These rules can now be summarized in terms of the operators $\mathfrak{L}$ and $\mathfrak{M}$. These operators transform the sequence of page references, one at a time, into a sequence of minimum memory capacities (or OPT stack distances), as shown in Fig. 12.

The operator $\mathfrak{L}$ transforms the sequence R of page references into the sequence L of LRU positions. It is defined as follows:

- If the page referenced next is not in the LRU stack, put the page on the top of the stack, compute the number of pages n in the stack, generate $(n-1)$ as output, and return.
- Else find the page at the kth position in the LRU stack, move the page to the top of the stack, generate $(k-1)$ as output and return.

Figure 11 Stack transition for Figure 10.

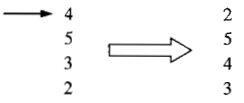
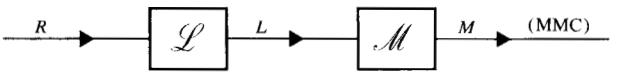


Figure 12 $\mathfrak{L}$ and $\mathfrak{M}$ operators.



The operator $\mathfrak{M}$ transforms the sequence L of LRU positions into a sequence M of minimum memory capacities. It keeps past output values in the P stack (a modified LRU stack), and it is defined as follows:

- If the next input ℓ is zero, return.
- Else
  - * If the ℓth position in the modified stack is empty, put the numeral $(\ell+1)=p$ on the top, generate p as output, and return.
  - * Else find the smallest value p at or below the ℓth position on the modified stack.
    - ◦ If p is at the ℓth position, move p to the top, generate p as output, and return.
    - ◦ Else find the smallest value p' above p but at or below the ℓth position; exchange p and p' in the modified stack, and reenter the previous step.

Application of these operators to part of our example reference string is shown in Table 1. Because the first reference is to page A, which is not in the stack, page A is put at the top of the stack. Since the number of pages in the stack is $n=1$, generate $n-1=0$; thus ℓ is zero and there is no output produced by operator $\mathfrak{M}$. The procedure is generally similar until we reach the second reference to D. Since D is now in the LRU stack at the second position, $k=2$, and $k-1=1$ is generated as output. With the P stack as shown for the fifth reference (to page E), the smallest value at or below the first posi-

Table 1 Application of $\mathscr{L}$ and $\mathscr{M}$ operators.

Nr.	*In*	$\mathscr{L}$	*k*	*LRU stack*	*n*	ℓ	$\mathscr{M}$	*P stack*	*Out*
1	A	Not in LRU stack, so put on top	–	A	1	0	$\ell = 0$, return	–	–
2	B	"	–	B A	2	1	1st position empty in *P* stack	2	2
3	C	"	–	C B A	3	2	2nd position empty in *P* stack	3 2	3
4	D	"	–	D C B A	4	3	3rd position empty in *P* stack	4 3 2	4
5	E	"	–	E D C B A	5	4	4th position empty in *P* stack	5 4 3 2	5
6	D	D is at $k = 2$ in LRU stack	2	D E C B A	5	1	2 is smallest value at or below 1st position. 3 is the smallest above it, so exchange them 5 4 2 3 Now 2 is again smallest, but 4 is the minimum above 2, so exchange them 5 2 4 3 etc.	 2 5 4 3	 2

tion is $p = 2$ at the fourth position. The smallest value above the fourth position, p', is 3; exchanging p and p' results in the stack shown in Table 1. Again, $p = 2$ is the smallest value below the first, and $p' = 4$ is the smallest value above p, so p and p' are again exchanged. The procedure is continued until the P stack for the second reference to D appears as shown in the table.

It is easy to show that the last two steps are identical to a single pass of a "bubble" sort [9] (strictly pairwise exchange sort). The operator $\mathfrak{M}$ then can be loosely described as follows:

"Find the minimum value of p in the lower part of the P stack, i.e., elements at and below the ℓth position, by performing a single pass of a bubble sort; then move p to the top."

3. Implementation considerations

As mentioned previously, a number of considerations indicate the possibility of a hardware implementation of the multivalued MIN algorithm that can be used in a production environment. Indeed, patent applications for implementation schemes have been filed.

First, some general machine requirements are suggested. For nontrivial experimentation, a paging machine with a viable page replacement algorithm is needed to accommodate programs larger than available real memory space.

The most important characteristic of the new algorithm is of course that it generates the MMC string in a single pass. Additionally, many of the steps of the $\mathfrak{M}$ and $\mathfrak{L}$ operators can be accomplished in parallel, using, for example, associative memory techniques. This parallelism, combined with decreasing computer component costs, makes feasible operation at essentially full computer speed. Other techniques that can contribute toward making a hardware implementation practicable by increasing the efficiency with which storage and processing resources are used are considered in this section.

On-line processing of the page reference string as it is produced eliminates the need for its storage. All of the information that need be stored is a table having a number of elements equal to the size of the program being measured.

It is further proposed that the table of the device used for implementing the algorithm consist of two parts, the first or upper part in fast logic, having as many entries as there are page frames in main memory. If, as usual, the program is larger than main memory, the remaining second (overflow) part of the table could be stored in some slower device, such as a protected area of main memory itself. At the time of a page fault, the contents of the two parts of the table could be updated to reflect the new memory contents.

It can be shown that, if a particular algorithm, LRU, is used to do the actual page management of the machine, a partitioning of the table into real and external parts is possible. Only pages in main memory must have corresponding entries in the device stack, updated at the rate of processing.

Also, given a fast LRU device, needed for on-line replacement anyway, only the operator $\mathfrak{M}$ described earlier need be built, and it could be an add-on device. Implementation of a device, operating at CPU speed and describing the entire virtual page space, would not be technologically feasible. However, with the relaxed requirement of having to represent only pages occupying main memory page frames, more than an order of magnitude smaller in number, the device could be built.

Hardware implementation of the LRU replacement algorithm has been proposed [10]. We simply assume that the device is a functional part of the (host) paging machine, and that its output can be tapped to derive the needed information to drive the add-on $\mathfrak{M}$ box, which, in turn, generates the MMC string. Operation of the two devices would then be as follows:

- The operator $\mathfrak{L}$ dynamically orders all page frames to reflect the order of past references.
  - * If the page referenced next is in the LRU stack, its stack position k is presented to the modified $\mathfrak{M}$ operator.
  - * Else a page fault is generated. The contents, if any, of the page frame with the highest LRU value (at the bottom of the stack) are pushed, the required I/O operation is initiated, the page referenced next is placed on the top of the stack, and a special program, held in memory to update all of the P stack information, is invoked.
- The modified $\mathfrak{M}$-operator processes the upper part of the partitioned P stack. It is almost identical to the algorithm presented earlier for the contiguous single stack except for the fourth step, finding the smallest p value. Since only a partial description of the stack is held in the fast upper part, a complement search is executed.
- In the fourth step, not p but q, the smallest missing integer above the ℓth position in the stack, is moved to the top and presented as output.

As a result of these steps, one of two conditions can exist: either p is already in the upper stack or it is not. If it is, then $p = q$; hence the complete and the partitioned schemes coincide. The necessary update is limited to the upper portion, and the algorithm is correctly executed.

If p is not in the upper stack, the generated missing q value is put on the top of the upper stack anyway. This results in a push-down operation (observing the rules of the algorithm) and in the removal of a value from the stack. Hence a new value is added to the top, while an old value drops out. Correctly, the new value should have been obtained from the lower part of the stack, while the removed value should have been pushed into the lower part. This neglect results in one duplicate value and one entirely missing value.

Such dual errors occasionally occur. Sooner or later, however, a page exception occurs and, as said before, a program is invoked that brings the two parts into alignment again by correcting, one by one, the pairs of errors. The program logic is as follows:

- Form the smallest integer p that is not represented in either part.
- Find in the lower part the smallest integer q that is smaller than p.
- If there is no element smaller than p and above q in the lower part, replace q with p, i.e., discard q, and return.
- Else find in the lower part the smallest integer q' that is smaller than p and is above q; exchange q' and q and reenter the previous step.

The process terminates when there are no more missing integers. It is easy to show that at this point all duplicates are also eliminated by discarding q's; the two parts become consistent, having unique p values. It is important though that this algorithm proceed from smaller to larger missing integers.

After the alignment of the parts, the p value of the page causing the page exception must be found. This is done by essentially executing the $\mathfrak{M}$ operator, the only difference being that the output found has to be put on

the top of the upper part directly. The resulting push-down then moves the element from the bottom of the upper part to the top of the lower part, inducing the execution of the last three steps of $\mathfrak{M}$ there. The fast upper part operates at the rate of processing; on the other hand, there is always sufficient time to update the lower part, since a page exception is associated with a slow I/O operation.

A device can be designed such that shifting bits, rather than counters, represent stack values in order to increase speed – simple logical operations can be used, instead of slower arithmetic operations.

However, even with very fast logic, references may be generated too fast for the device to keep abreast. At least two alternatives present themselves: slowing down the machine or using one of the slower members of the compatible IBM System/370 family of computers for measurement and testing purposes. In addition, repetitions can be eliminated from the reference string by a device, such as a look-aside associative memory, that buffers one, or even two (for data and program), page names and propagates a next element to the device only if there is a change of buffer contents. By doing this the machine and the device become more asynchronous, and this, in turn, may call for further buffering of the now repetition-free reference string. Interlocks can also be employed to inhibit program execution in the rare case of buffer overflow and a resulting lost reference. In any event, it seems certain that the extraction of the minimum memory capacity string can be accomplished while the machine executes a program at essentially full speed, as opposed to the 1:20 or more slow-down associated with interpretive program execution when the reference string first has to be recorded.

In summary, with an already implemented LRU box, our device becomes an add-on instrument for extracting the MMC string.

4. Program behavior

The information extracted by the multivalued MIN algorithm and the way that this information is structured in the P stack reveal some interesting program properties and form a basis for the development and evaluation of better algorithms for assigning page frames to programs in a multiprogramming environment. Some of these ideas are centered on the notion of output classes, as defined below.

Recall the reference string ABCDE and its P stack after the reference to E. If the next, sixth, reference is to any page but E, then 2 is generated as output by the multivalued MIN algorithm. Taking the same example after the seventh reference, ABCDEDB, the P stack read from the top is 3254, meaning that only three distinct outputs are possible: 1 (for repetition), 2 and 4.

In order to introduce the idea of equivalence classes, we define, at any point of the reference string, the MMC class C_j as the collection of pages whose next (potential) output is j. As usual $|C_j|$ means the cardinality of set C_j.

From the previous sections it is obvious that the LRU replacement algorithm is also a page classification algorithm. It dynamically classifies the program pages into as many classes as there are distinct pages in the program (i.e., distinct elements in the reference string). As a result, at any time there is one and only one element in each class.

In contrast, our algorithm classifies the pages, on the average, into a lesser number of classes by grouping candidates of optimum choice for a particular memory size. As the referencing pattern evolves, the average size of classes – or the level of uncertainty with respect to the optimum choice – fluctuates according to the particular pattern of (re-)referencing elements. The removal by a reference of a class member eliminates the chance of its peers being retained in the same size memory; their chances are lessened. In fact, one can envision an implementation of the $\mathfrak{M}$ operator applied to the LRU stack via pointers, separating the pages into MMC classes.

By inspecting the class distribution it can also be shown that the efficiency (as described earlier) of LRU with respect to MIN has a nonzero lower bound for any input string; it is $1/M$, where M is the memory size. This is not true for any other algorithm: the RAND replacement algorithm [3] has, for example, zero as lower bound.

There exist strings that cause MIN to mimic LRU. Consider the following program, referencing a sequence of strictly nested localities:

ABCDEDCBCDC

It can easily be verified that at the end of the string the 5-entry P stack contains, (1),2,3,4,5, which is identical to the complete order enforced by LRU.

One may wish to consider this strict nestedness to be a strong locality. On the other hand, a sequential referencing of distinct pages (without re-referencing) produces a P stack grouping all but one element into class 2. This sequential referencing phase, as in the example string used in this paper, corresponds to the acquisition of pages for a new locality, before re-referencing. One could venture the statement that an LRU-like behavior would identify a well established locality, while a large class 2 implies drifting from one locality to another (collecting "fresh" pages).

It can also be shown that in the case of cyclically re-referencing the same group of pages, as in a program loop, the average size of class 2 is about half way between the two extrema mentioned above.

In a multiprogramming scheme, a device could, for example, constantly update a register to contain

$$\sum_{j=2}^{n} |C_j| j,$$

where n is the total number of pages of a program. If this sum is low (close to $(n-1)2$), the program must have been in transition. If it is close to the maximum (LRU) value of $(n-1)(n+2)/2$, then a locality must have been established. Having one register for each program, this could be a basis for a dynamic multiprogramming scheme, helping to detect the relative memory space requirements of the program mix. There is, in general, a growing need for more on-line detection of program behavior to improve system operation.

Another interesting phenomenon is the regularity of class sizes. Class 1 has obviously only one member and is never empty. Its only member is the most recent element in the reference string. The next reference destroys this membership and forces the element from class 1 into class 2. This implies that class 2 is never empty either (except for a single-page program).

Consider now an element α of class C_j. If the next reference is made to a member of another class than C_j, α will stay in C_j. If, however, another element, not α, is referenced in C_j, then α's membership changes to class C_k, $k > j$; in fact the referenced element becomes itself the sole element of class 1 and, again, class C_j becomes empty.

For the nonrepetitious input string, it is now obvious that elements (pages) of the program go through class memberships in a cyclic fashion. Starting after class 1, the membership changes through an increasing sequence of integer values and eventually returns to 1 again.

Since every element goes through class-2 membership for each of its occurrences in the input string, the size of class 2 increases by one every time a reference is made to an element whose class is other than 2. If, however, a class-2 element is referenced, this class collapses and becomes the holder of one element only, namely the one dropping out of class 1. In contrast, any class C_j $(j > 2)$ increases in clusters by the collapse of, for example, C_{j-1}. On the other hand, class C_j loses all its members by a reference to any element in it and creates (or increases the size of), for example, class C_{j+1}.

A further observation is that the output string is constrained and not all sequences of integers are possible output strings. This becomes obvious when we notice that, at any point in the output string, in general, integers are not in a strictly increasing order in the stack. From earlier reasoning, 2 is always a candidate (and so is 1 for the repetitious case). But some classes may be empty while other classes may have a multiplicity of elements.

There has not been enough time since the discovery of the approach presented in this paper to study this constrainedness of the output. Some aspects are, however, obvious. Suppose that pages are identified by numerals as symbols. Even then the input and output strings are not interchangeable; since any numeral should be a possible next input but not every numeral is a possible next output, many input strings map into a single output string. If the inverse to the presented algorithm exists, then it will produce from the output string an input string that is a representative of a class of reference strings having the same space/time behavior.

An additional obvious constraint is that the highest integer (class label) cannot exceed the number of distinct elements already represented in the input. Also, if class C_j is empty but C_{j+1} is not, then there exists a class i, $i < j$, such that it has more than one element. Furthermore, no runs (repetitions) are possible in the output strings since the occurrence of, say, p makes this class empty, and before p can occur in the output it has to be replenished. Class 2 is exempt and can have runs in the output since it is fed by each reference from class 1, as seen earlier.

It seems that the contents of the P stack, its use to construct the output string, and the interpretation of the table updating as state transitions offer an entirely different view of the memory referencing behavior of programs.

5. Summary

This paper described an algorithm that generates from a page reference string the sequence of minimum memory capacities required to execute a program. The primary feature of the algorithm is that the output string is produced for the range of memory sizes in a single pass.

The algorithm was first described in terms of numeral matrices, which permitted its correspondence to the MIN algorithm to be demonstrated and also provided a visual means of comparing minimum paging with that obtainable using LRU replacement. A stack processing scheme to reduce overhead was described for generating the MMC string, the description demonstrating the equivalence of the string to the sequence of OPT stack positions. (A formal proof of this equivalence is provided in the Appendix.)

A scheme was outlined for implementing the new algorithm in such a way as to reduce the burden on computing system processing and storage resources. Thus the output formerly obtainable from the two-pass OPT algorithm can now be produced essentially at normal processing speed. Also, some of the possibilities of using the data generated by the new algorithm to gain new insight into program addressing characteristics were discussed. Both of these topics suggest numerous avenues for further exploration.

Acknowledgments

This work was motivated primarily by the curiosity of the students at the University of California at Berkeley. Many thanks are also due to G. S. Shedler, who first recognized the program behavior implications, to D. R. Slutz for his invaluable counter-example, and to R. A. Nelson for illuminating alternative views.

References

1. R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal* **9**, 2, 78 (1970).
2. P. J. Denning, "Virtual Memory," *Computing Surveys* **2**, 3, 153 (September 1970).
3. L. A. Belady, "A Study of Replacement Algorithms for Virtual Storage Computers," *IBM Systems Journal* **5**, 2, 178 (June 1966).
4. F. R. A. Hopgood, *Compiling Techniques*, MacDonald, London, 1970, 96–99.
5. D. Gries, *Compiler Construction for Digital Computers*, John Wiley and Sons, New York, 1972.
6. In 1970, J. Gecsei of IBM Systems Development Division, San Jose, developed a one-pass version of OPT yielding the approximate space-time behavior of programs.
7. L. P. Horwitz, R. M. Karp, R. E. Miller, and S. Winograd, "Index Register Allocation," *J. ACM* **13**, 1 (January 1966).
8. U. S. Patent 3,577,185, "On-line System for Measuring the Efficiency of Replacement Algorithms," issued to L. A. Belady.
9. *Sorting Techniques*, IBM Data Processing Techniques, Form NR. C20-1639-0, 12–13, IBM Corporation, White Plains, N.Y.
10. W. F. Beausoleil, D. T. Brown, and B. E. Phelps, "Magnetic Bubble Memory Organization," *IBM J. Res. Develop.* **16**, 6, 587 (November 1972).

Appendix: Proof of equivalence to OPT

The correctness of the original two-pass OPT algorithm was proved in [1]. Our multivalued MIN algorithm can be considered to be a one-pass OPT algorithm, as we mentioned previously. In this Appendix, we prove that the numerals constituting the MMC string are in fact OPT stack positions. In order to do this, we first review briefly both algorithms from a slightly different point of view, introducing at the same time some additional notation.

• *One-pass OPT algorithm*

Recall that, for the multivalued MIN algorithm, two stacks are maintained. The LRU stack is a list of the pages that have been referenced with the most recently referenced page on top of the stack. In general, page α is above page β if and only if α has been referenced more recently than β.

The P stack is a list of integers, each integer representing the potential minimum memory size required to include the page with which the integer is associated. When a page is referenced, its position in the LRU stack is found. The smallest integer in the P stack that is at or below the position of the page in the LRU stack is, we assert, the OPT stack distance.

The LRU stack is updated by moving the page referenced to the top of the stack and then pushing down one position each page above the original position of the referenced page.

The P stack is updated as follows. The value at the position of the referenced page is removed from the stack and saved. All items between the vacated position and the second position from the top of the stack are moved down one position. The saved value is compared with the value in the next lower position in the stack. The larger of these values is placed in this position, and the smaller value is saved. The process is continued until the bottom of the P stack is reached. The final saved value (the OPT stack position or MMC) is placed in the second position of the P stack.

Note that the P stack as described here differs from our earlier P stack in that a 1 is always in the top position. Thus the number of entires in the LRU stack and P stack are always the same.

An alternative description of this update procedure is as follows: Let k be the LRU stack position of the referenced page. All items on the P stack between 2 and $k-1$ inclusively are marked. The next item marked is the one that has the minimum value from among the items that are at or below position k and above the last marked item, if any. This process is continued until the item at position k is marked. The marked item in the lowest position is the output value. It is put in the second position of the P stack. All other marked items are shifted to the next lower marked position. We will show the output to be the OPT stack position, as defined below.

• *Two-pass OPT algorithm*

The two-pass OPT algorithm may be summarized as follows. On a first pass through a page address trace, the algorithm produces the forward distance string by creating for each page referenced the number of distinct pages referenced before that page is re-referenced. Thus, let $R = x_1, x_2, \ldots, x_N$ be the reference string, and let $F = F_1, F_2, \ldots, F_N$ be the forward distance string. The two-pass OPT algorithm is a stack algorithm that uses a priority list created from the forward distance string to determine the new OPT stack positions after a page has been referenced. Thus at time t let Q_t be the OPT stack and L_t be the priority list. *If* x_{t+1} is the next page referenced, its position k in Q_t is the OPT stack distance and is placed in the output string. The referenced page is moved to the top of the Q-stack and displaces the page already there. The items above position k are rearranged as follows: The priority for the displaced page and the page in the next position are compared. The page with higher priority

(lower priority number) is placed in this position, while the other page is displaced. This process continues until the next page position is k; then the last displaced page is placed in position k.

Priority list L is updated as follows. All priorities that are less than or equal to the forward distance F_{t+1} are reduced by one, and the priority for the referenced page x_{t+1} is F_{t+1}.

• *Example of two-pass OPT*

Let the given sequence of page references be ABCDED BCB. The requirement that an arbitrary priority be given to pages that are not referenced again can be met by adjoining to the end of this string the set of referenced pages in some order (e.g., alphabetical order as in [1]). Thus the given page reference string is extended to the string ABCDEDBCB (ABCDE). The forward distance string for the original string is found to be 5 4 4 2 5 4 2 3 2. The priority lists are given in Fig. 13.

For example, we obtain the second from the last column of the priority list in Fig. 13 (2 1 3 4 5) from the previous column (3 2 1 4 5) by observing that the referenced page C has a forward distance of 3. Then the entries 3 and 2 are reduced by 1, and the entry 1 is changed to 3 to obtain the required updated priority list.

The OPT stack positions, displayed in Fig. 14, are obtained by using this priority list. For example, the stack positions in the third from last column (5 1 4 2 3) are changed when C is referenced as follows:

The referenced page C is given stack position 1. Since C occupied stack position 4 and is changed to stack position 1, the page formerly in position 1 must change and the pages in positions 2 and 3 may also change. Thus B in position 1 and D in position 2 have their priorities compared. Since B has a priority of 2 and D has a priority of 4, B takes stack position 2 while the priority of D is compared to the priority 5 for E, the page in position 3. Then D takes stack position 3 and E drops down to position 4, the original position of the referenced page.

• *Partially filled numeral matrix*

We have shown informally in section 2 that the MMC-values can be generated in one pass and claimed that the MMC values are, in fact, the OPT stack distances. In our numeral matrix algorithm, however, we maintain a sequence of OPT stacks only partially filled in, as opposed to the two-pass OPT algorithm, which maintains stacks for all times up to the present time, because the first pass supplies all necessary information about the future.

In Table 2, the sequence of partially filled in numeral matrices H_t are given for the above reference string. To

Figure 13 Two-pass OPT priority list updating.

	A	B	C	D	E	D	B	C	B	(A	B	C	D	E)	
	5	4	4	2	5	4	2	3	2						forward distance string
A	5	5	5	5	4	3	3	2	1						
B		4	3	3	2	1	2	1	2						
C			4	4	3	2	1	3	3						priority lists
D				2	1	4	4	4	4						
E				5	5	5	5	5							

Figure 14 OPT stack updating.

	A	B	C	D	E	D	B	C	B	
A	1	2	3	4	5	5	5	5	5	
B		1	2	2	3	3	1	2	1	
C			1	3	4	4	4	1	2	OPT stacks
D				1	2	1	2	3	3	
E					1	2	3	4	4	
	∞	∞	∞	∞	∞	2	3	4	2	OPT stack positions (outputs)

conform with the notation for the two-pass OPT algorithm, the output associated with a first reference to a page is the symbol ∞. We also list the available stack position sets A_t and the unassigned page sets B_t, which are defined formally in the following text. For A_t and B_t only the nonempty columns are shown.

We presented earlier the matrix updating procedure, which is now given an interpretation using OPT-stack terminology. For example, when C is referenced at time 8, the row labeled C is filled for times 4, 5, 6, 7, and 8. For these times, the available stack positions (obtained from A_t for $t = 7$) are given by

t	4	5	6	7
				3
			2	2
	4	5	5	5
	3	4	4	4

Thus the sequence of entries 3 4 4 4 1 is made. For $t = 4$, the minimum, 3, of the available set $\{3, 4\}$ is assigned to C. For $t = 5$, the minimum of the set $\{4, 5\}$ is assigned to C, both 4 and 5 being greater than 3. For $t = 6$, the value 4 is assigned to C because it is the minimum of the set

$$\{k \mid k \geq 4 \text{ and } k \in \{2, 4, 5\}\}.$$

Similarly, 4 is assigned to C for $t = 7$. For $t = 8$, the stack position 1 is assigned to C.

This informal procedure now enables us to establish that the one-pass algorithm yields the same output as the two-pass OPT algorithm of [1].

Table 2 One-pass OPT numeral matrix and unassigned position and page lists.

Time t	*Page referenced* X_t	*Numeral matrix* H_t	*Unassigned position list* A_t	*Unassigned page list* B_t	*Output* MMC_t
1	A	A 1	ϕ(empty)	ϕ(empty)	∞
2	B	A 1 B – 1	2	A	∞
3	C	A 1 B – 1 C – – 1	3 22	B AA	∞
4	D	A 1 B – 1 C – – 1 D – – – 1	4 33 222	C BB AAA	∞
5	E	A 1 B – 1 C – – 1 D – – – 1 E – – – – 1	5 44 333 2222	D CC BBB AAAA	∞
6	D	A 1 B – 1 C – – 1 D – – – 1 2 1 E – – – – 1	2 455 3344 22233	E CCC BBBB AAAAA	2
7	B	A 1 B – 1 2 2 3 3 1 C – – 1 D – – – 1 2 1 E – – – – 1	3 22 4555 233444	D EE CCCC AAAAAA	3
8	C	A 1 B – 1 2 2 3 3 1 C – – 1 3 4 4 4 1 D – – – 1 2 1 E – – – – 1	4 33 222 2345555	B DD EEE AAAAAAA	4
9	B	A 1 B – 1 2 2 3 3 1 2 1 C – – 1 3 4 4 4 1 D – – – 1 2 1 E – – – – 1	2 344 2233 23455555	C DDD EEEE AAAAAAAA	2

• *Formal structure*

First we reinterpret, and extend, earlier notations. Our partially filled numeral matrix will now appear as a *partial function* H_t. Lists A_t and B_t, introduced in Table 2, will be formally defined. The symbol k is any stack position. A numeral string $S(\alpha)$ consists of a sequence $\{\gamma_i\}$ of integers, $t_\alpha < i \leq t$, where α is a page previously referenced at time t_α. The set of consecutive integers from a to b is denoted by $[a, b]$.

Let Σ_t be the set of pages referenced up to time t and let n_t be the number of pages in Σ_t, i.e., $n_t = |\Sigma_t|$.

Recall that a partial function from a domain D to N is a function defined on a subset of D to N. Thus in our case the partial function H_t is defined on a subset of $[1, t] \times \Sigma_t$ to the interval $[1, n_t]$, i.e., $H_t{:}[1, t] \times \Sigma_t \rightarrow [1, n_t]$.

These partial functions will be defined recursively. First define H_1 to be the function that assigns to $(1, x_1)$ the value 1, i.e.,

$H_1 = \{(1, x_1), 1\}$ or

$H_1(1, x_1) = 1$.

Now suppose H_t has been defined on a subset of $[1, t] \times \Sigma_t$ to $[1, n_t]$. Thus H_t is the partially filled numeral matrix. Let $A_t(i)$ be the complement of the set of entries of the ith column of H_t. Similarly $B_t(i)$ is the set of pages in Σ_t for which the ith column does not contain an entry. Formally, we have

$$A_t(i) = [1, n_t] - \{k | (\exists \alpha \in \Sigma_t) \text{ such that } H_t(i, \alpha) = k\}$$

$$B_t(i) = \Sigma_t - \{\alpha | (\exists k \in [1, n_t]) \text{ such that } H_t(i, \alpha) = k\}.$$

For any page α, we define the *immediate minimum string* (IMS) $\{\gamma_i|$ for $t_\alpha < i \leq t+1\}$, where $t_\alpha =$ the last time α was referenced. (If this is the first time α is referenced, $t_\alpha = 0$.) First define $\gamma_{t_\alpha} = 1$ and $\gamma_{t+1} = 1$. Then define γ_i for $t_\alpha < i \leq t$ by the formula

$$\gamma_i = \min\{k | k \geq \gamma_{i-1} \text{ and } k \in A_t(i)\}.$$

For a given α, the partial function $S_t(\alpha)$ is defined for all (i, α) such that $t_\alpha < i \leq t+1$ by the formula

$$S_t(\alpha)\ (i, \alpha) = \gamma_i.$$

Now if the reference $x_{t+1} = \alpha$, the partial function H_{t+1} is defined to be the extension of H_t by adjoining the IMS $\{\gamma_i | t_\alpha < i \leq t+1\}$. Thus

$$H_{t+1} = H_t \cup S_t(\alpha).$$

Using this definition of the sequence H_t, we can now prove the following theorem.

Theorem 1: If $H_t(i, \alpha) = k$, then k is the OPT stack position for page α at the time i.

This theorem results from the following reasoning.

Lemma 2: If H_t gives the OPT stack table at time t, then $S(\alpha)$ determines the OPT stack entries for $\alpha = x_{t+1}$ for all times since α was last referenced.

Proof: If α was not previously referenced, then its OPT stack position for all $i < t+1$ is ∞, and for $i = t+1$, its position is 1.

If α was last referenced at time t_α, then its stack position at that time was 1. The position of α at each time $t_\alpha < i \leq t$ is determined in the OPT algorithm as follows: If α was at stack position γ_i at time i, it will remain at position γ_i if γ_i has not been assigned to any other page at time $i+1$, i.e., γ_i is in the set $A_t(i+1)$. (Note that i as used here corresponds to τ as used in section 2.) Otherwise α will be displaced from position γ_i if γ_i has been assigned to some other page, i.e., $\gamma_i \notin A_t(i+1)$. In this case its stack position will be the first unassigned stack position that is greater than γ_i.

The OPT algorithm may be interpreted as follows: A page α whose stack position is γ_i at time i will remain at that position at time $i+1$ because

1. The stack position of the page referenced at time $i+1$ is less than γ_i (i.e., α is not challenged), or
2. If α is challenged, it has a higher priority than all challengers for its position. But this can happen only if γ_i has not been assigned to a page with higher priority.

Suppose α is at stack position γ_i at time i. Suppose that page β is referenced at time $i+1$ and is in position Δ_i: then if $\gamma_i > \Delta_i$, α will remain in position γ_i. The stack positions for all pages with higher priority than γ have already been determined (by the induction hypothesis). Thus α will be assigned the first available position that is $\geq \gamma_i$. A page cannot achieve a smaller stack position than its present value. In the contests for any given stack position, the positions for all pages with higher priority than α have already been assigned. Thus α may compete for all stack positions $\geq \gamma_i$ and will win the first contest in which it has a higher priority. But this will happen only for stack positions that have not been assigned. Thus the stack position for α at time $i+1$ is given by

$$\min\{k | k \geq \gamma_i \text{ and } k \in A_t(i+1)\}.$$

In the definition of the partial functions, we introduced the sets $A_t(i)$ and $B_t(i)$ for $i \leq t$. These sets will be used to establish the connection to the one-pass OPT algorithm. First we list some properties of $A_t(i)$ and $B_t(i)$.

1. $B_t(i) \subseteq B_t(i+1)$
2. $|A_t(i)| \leq |A_t(i+1)|$
3. $|A_t(i)| = |B_t(i)|$
4. $|A_t(i)| < |A_t(i+1)| \Rightarrow A_t(i+1) = A_t(i) \cup \{k\}$ for some k.
5. If $|A_t(i)| = |A_t(i+1)|$ then $A_t(i+1) = (A_t(i) - \{j\}) \cup \{k\}$ for some j and k.
6. (a) $A_t(t) = [2, n_t]$,
   (b) $B_t(t) = \Sigma_t - \{x_t\}$.

Thus, given H_t, the sequence of sets $A_t(i)$ and $B_t(i)$ are determined. If the page α is referenced at time $t+1$, i.e., $x_{t+1} = \alpha$, then H_{t+1} is constructed by adjoining $S_t(\alpha)$ to H_t. The resulting sequences $A_{t+1}(i)$ and $B_{t+1}(i)$ as obtained from H_{t+1} can also be obtained from the sequences $A_t(i)$ and $B_t(i)$ using formulas 7 and 8 below if α has been previously referenced and formulas 9 and 10 below if α has not been previously referenced. Formulas 6a and 6b give the values for $A_{t+1}(t+1)$ and $B_{t+1}(t+1)$.

If α has been previously referenced, then $\Sigma_{t+1} = \Sigma t$ and $n_{t+1} = n_t$.

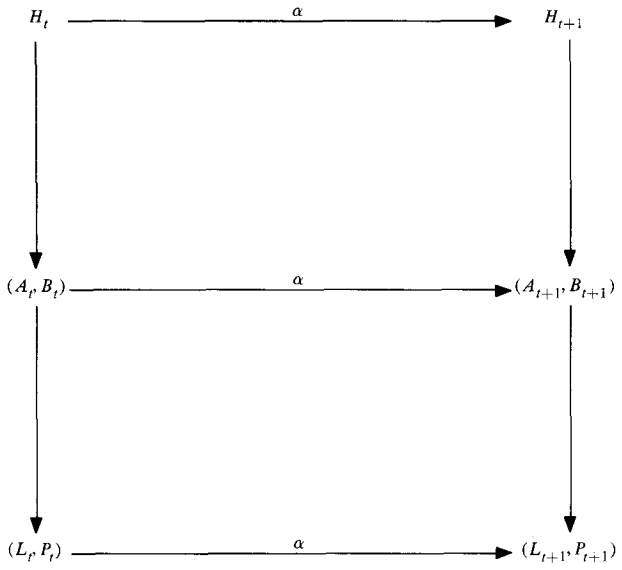


Figure 15 Effects of a reference to page α.

7. $A_{t+1}(i) = A_t(i) - \{\gamma_i\}$.

8. $B_{t+1}(i) = B_t(i) - \{\alpha\}$ for $t_\alpha < i \leq t$.

   If α has not been previously referenced, then $\Sigma_{t+1} = \Sigma_t \cup \{\alpha\}$, and $n_{t+1} = n_t + 1$.

9. $A_{t+1}(i) = A_t(i), \quad 1 \leq i \leq t$.

10. $B_{t+1}(i) = B_t(i), \quad 1 \leq i \leq t$.

The above formulas are easily demonstrated and their proofs are left to the reader.

• *Definition of L and P stacks*

The L_t and P_t stacks are obtained directly from the sequence of sets $A_t(i)$ and $B_t(i)$ as follows. First we introduce the sequence of times t_i, $i = 1, \cdots, n$, where $n = n_t$. Time t_i is defined as the first time that i pages were not assigned, i.e., $t_i = \min \{s \mid |B_t(s)| = i\}$ for $i = 1, \cdots, n-1$ and $t_n = t + 1$. Also define $B_t(t+1) = \Sigma_t$. Then we let α_i be the page that is in $B_t(t_i)$ but not in $B_t(t_i - 1)$. Thus

$$\alpha_i \in B_t(t_i) - B_t(t_i - 1) \text{ for } i = 1, \cdots, n.$$

Note that the sequence of pages $\alpha_1, \cdots, \alpha_n$ are in LRU order with α_1 being the least recently referenced and α_n the most recently referenced. Note also that α_n is the page referenced at time t. For each $i = 1, \cdots, n-1$, let $S(\alpha_i)$ be the string adjoined to the partial function H_t after strings $S(\alpha_1), \cdots, S(\alpha_{i-1})$ have been adjoined and define $S'(\alpha_i)$ to be the string $S(\alpha_i)$ truncated at time t. Thus

$$S'(\alpha_i) = \{ (j, \alpha_i, \gamma_j) \mid (j, \alpha_i, \gamma_j) \in S(\alpha_i) \text{ and } t_i \leq j \leq t \}.$$

Let X_t be set $[1, t] \times \Sigma_t \times [1, n]$ and $G_t = X_t - H_t$. Then

$$G_t = \bigcup_{i=1}^{n-1} S'(\alpha_i).$$

Note that the strings $S'(\alpha_i)$ are the $n-1$ primitives shown in Fig. 8.

The set G_t when written as the indicated union of primitives corresponds to the rearrangement of the entries in the complement of the numeral matrix into primitive strings.

Define P_t to be the set of integers $[1, n_t]$ in the order induced by the strings $S'(\alpha_i)$. Thus

$P_t(1) = 1$.

$P_t(j) = k$ if and only if

$(t, \alpha_{n-j+1}, k) \in S'(\alpha_{n-j+1})$ for $j = 2, \cdots, n$.

Thus P_t is the P stack defined earlier and marked in Fig. 8.

Also define L_t to be the set of pages Σ_t in the order induced by the sequence $\alpha_1, \cdots, \alpha_n$, i.e., in LRU order. Thus

$L_t(j) = \alpha_{n-j+1}$ for $j = 1, \cdots, n$.

L_t is the LRU stack. With these definitions, we now have the following lemmas.

Lemma 3: If $\alpha_K \in \Sigma_t$ is referenced at time $t+1$, the output γ_t is given by the formula

$$\gamma_t = \min A_t^K(t),$$

where

$$A_t^K(t) = \{k \mid \exists j (t, \alpha_j, k) \epsilon S'(\alpha_j) \text{ and } j \leq K\}$$
$$= \{P_t(j) \mid j \geq n - K + 1\}.$$

Note that $A_t^K(i)$ is the set of elements of $A_t(i)$ that are at level K or lower, i.e., whose index is $\geq K$. In fact, it can be shown that the IMS for the referenced page α is calculated by selecting the minimum element in $A_t^K(i)$ for $t_K < i < t+1$. Thus we have the formula.

$$\gamma_i = \min A_t^K(i) \text{ for } t \geq i \geq t_K,$$

which is easily proved by induction.

The next theorem asserts that the updating of stacks L_t and P_t are in step with the updating of the sets $A_t(i)$ and $B_t(i)$ for $i \leq t$. Note that A_t and B_t denote the sets of lists $A_t(i)$ and $B_t(i)$ for $i < t+1$.

Theorem 4: If P_t and L_t correspond to the pairs A_t and B_t, and $\alpha_K \in \Sigma_t$ is the next page referenced, then the stacks P'_{t+1} and L'_{t+1} updated by the one-pass OPT procedure are the same as the stacks P_{t+1} and L_{t+1} corresponding to the pairs A_{t+1} and B_{t+1}.

This theorem follows from the fact that the transition from $A_t(i)$ to $A_{t+1}(i)$ is effected by removing the element

γ_i from $A_t(i)$ and reordering the remaining elements using the updating algorithm for the P stack.

This is made precise by the following definitions and lemma. Let $P = \langle P_1, \cdots, P_N \rangle$ and $Q = \langle q_1, \cdots, q_N \rangle$ be ordered sets of integers. We say that $P \leq Q$ if P and Q satisfy the following conditions for $i = 1, \cdots, N$ and $j = 1, \cdots, N$:

1. $P_i \leq q_i$;

2. IF $p_i \leq q_j$ and $j \leq i$, then $q_i \leq q_j$.

The set $P' = P - \{p\}$, where $p = \min P$ is said to be *canonically reordered* if the elements p_i' of P' are obtained from P by the recursive formulas

$t_1 = P_1$;

$t_{i+1} = \min \{t_i, p_{i+1}\}$ for $i = 1, \cdots, N-1$;

$p_i' = \max \{t_i, p_{i+1}\}$.

Lemma 5: Let $P' = P - \{p\}$ and $Q' = Q - \{q\}$, where $p = min\ P$ and $q = min\ Q$. If P' and Q' are canonically reordered and $P \leq Q$, then $P' \leq Q'$.

This lemma is proved by a straightforward computation. It is used to construct an inductive proof of Theorem 2.

These theorems establish the equivalence of the one-pass OPT to the two-pass OPT algorithm. This follows because for any input the same output is obtained, and the updated stacks P'_{t+1} and L'_{t+1} correspond to the partial function H_{t+1}.

The above discussion may be summarized by means of Fig. 15. The horizontal arrows indicate that when page α is referenced at time $t+1$, the numeral matrix H_t, the pair of sequences of sets (A_t, B_t), and the pair of stacks (L_t, P_t) are transformed to the indicated objects. We have also shown how to construct the pair (A_t, B_t) from H_t and (L_t, P_t) from (A_t, B_t). The results of the above lemma may be stated as: The transformation from H_t to (L_{t+1}, P_{t+1}) is independent of the path taken in Fig. 15.

Received March 20, 1973

L. A. Belady is located at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York 10598. F. P. Palermo is located at the IBM Research Division Laboratory, Monterey and Cottle Roads, San Jose, California 95114.