Information Processing Letters 26 (1987/88) 71–75
North-Holland

19 October 1987

A HALVING TECHNIQUE FOR THE LONGEST STUTTERING SUBSEQUENCE PROBLEM

Andranik MIRZAIAN

Department of Computer Science, York University, North York, Ontario, Canada M3J 1P3

Communicated by E.C.R. Hehner
Received 4 September 1986
Revised 6 March 1987

This paper presents an optimal linear-time algorithm to solve the *longest stuttering subsequence problem.* A suitable variation of the recently developed *halving method* is used to achieve this result. This demonstrates yet another application of the halving method and gives an indication that the method can be considered as an *algorithmic paradigm.*

Keywords: Halving method, stuttering subsequence

1. Introduction

The *halving method* was recently developed in order to design a linear-time algorithm for the *optimum offset* problem in *VLSI river routing* [6,7]. The best previously known algorithms to solve this problem had an O(n log n) running time [2,11]. This method was also applied to obtain an O(n) time algorithm for the *selection* problem on $n \times n$ matrices with sorted rows and columns [8]. (Some previous solutions to this problem may be found in [3,5].) This problem has numerous applications in statistics [4,9].

In this paper, we intend to show yet another application of the halving method. This gives an indication that the method has a wide range of applicability and may be considered an an *algorithmic paradigm.*

The contents of this paper is organized as follows. Section 2 describes the halving method in its generality. Section 3 defines the *longest stuttering subsequence* problem. Section 4 describes a linear time solution of the problem based on the halving method. Section 5 makes some concluding remarks.

* This work was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC) under Grant No. A5516.

2. The halving method

The halving method may be described as follows. Suppose we are given a sequence $x = x_1 x_2 \ldots x_n$ and we need to compute an integer function f(x). Let x^o, called the *odd half* of x, be the subsequence of x comprising odd indices. That is, $x^o = x_1 x_3 x_5 \ldots$. We may similarly define x^e, called the *even half* of x, to be the subsequence of x comprising even indices. We notice that the length of each of sequences x^o and x^e is about half the length of x. We recursively compute $f(x^o)$, and we then use $2f(x^o)$ as an approximation to $f(x)$. A final adjustment will transform the approximate solution to the exact solution.

Some variations of the above general description are possible. One such example is the *Shellsort* algorithm [10]. Another example is, instead of using $2f(x^o)$ as an approximation to f(x), to use some combination of $f(x^e)$ and $f(x^o)$. This line of approach can be seen in the Fast Fourier Transform [1]. One difference is that in the latter case we need two recursive calls instead of one. For

efficiency reasons this should be avoided if possible. The halving method can also be generalized from one-dimensional to multi-dimensional sequences, such as matrices (see, e.g., [8]). In Section 4 we shall see another variation of the technique used.

3. The longest stuttering subsequence problem

Suppose $a_1, a_2, a_3, \ldots, a_m$ are given symbols (not necessarily distinct), for some $m \geq 1$. We define a^i to denote the sequence $a_1^i a_2^i a_3^i \ldots a_m^i$. That is, a^i is the sequence formed by i repetitions of a_1, followed by i repetitions of a_2, followed by i repetitions of $a_3, \ldots$, followed by i repetitions of a_m.

We are also given a sequence $x = x_1 x_2 x_3 \ldots x_n$ of length n. The problem is to find the maximum i such that a^i is a subsequence [1] of x. (Note that this maximum i is at most n/m and also a^0 is the empty sequence which is obviously a subsequence of x. Therefore, the problem is well defined. Also, from now on we shall denote a^1 by a.) We shall denote this maximum i by maxi(x, a). So the problem is, given x and a, to compute maxi(x, a).

3.1. Example. Suppose a = 012 and

x = 211010021101121102221020.

Then, maxi(x,a) = 4, since from the underlined positions in x we see that

$x_4 = x_6 = x_7 = x_{11} = 0$,

$x_{12} = x_{13} = x_{15} = x_{16} = 1$,

$x_{18} = x_{19} = x_{20} = x_{23} = 2$,

and no subsequence of x is equal to a^5.

In the next section we shall develop a linear-time algorithm, based on a suitable variation of the halving method, to solve the problem. We suspect that the method may be applied to a more general case of the basic problem stated above. However, we have avoided being exhaustive on this subject.

[1] A *subsequence* of x does not have to be a contiguous portion of x.

4. The algorithm

In order to solve the longest stuttering subsequence problem, we first need to develop a boolean function scan(x, a, i). This function returns *true* if and only if a^i is a subsequence of x. The "scan" function will then be used within the main algorithm.

4.1. The scan function

This function is quite straightforward as shown in Algorithm A. It linearly scans through the sequence x, 'collecting' the symbols of a^i as they are detected. The notation |x| denotes the length of the sequence x. As can be seen, this function takes O(n) time to compute.

Algorithm A

```
function scan(x, a, i);
var t; j; count;
begin
  if i = 0 then return true;
  t := 1; count := 0;
  for j := 1 to |x| do
    if x_j = a_t then begin
      count := count + 1;
      if count = i then begin
        if t = m then return true;
        t := t + 1; count := 0
      end
    end;
  return false
end.
```

4.1. Theorem. *The function* scan(x, a, i) *returns true if and only if* a^i *is a subsequence of* x, *and it does so in* O(n) *time.*

The proof of Theorem 4.1 is obvious.

4.2. The main algorithm: First attempts

The problem is to compute the integer function maxi(x, a). That is, to compute the maximum integer i such that scan(x, a, i) is true. In what follows, we shall suggest a number of ways to achieve this before describing our final solution based on the halving method.

Our first, and most naive, solution is a *linear search* on i. By the problem statement, we know that the maximum i is within the integer range 0..⌊n/m⌋. We may try successive values of i in that range to find the maximum i such that scan(x, a, i) is true. One such a linear-search algorithm is shown in Algorithm B. This algorithm takes $O(n^2/m)$ time, since at most n/m values of i are examined, and for each value of i the "scan" function is invoked which takes O(n) time.

Algorithm B

```
function maxi(x, a);   {a linear search algorithm}
var i;
begin
  i := 0;
  while scan(x, a, i + 1) do i := i + 1;
  return i
end.
```

Our second solution is a *binary search* on i. This algorithm, shown as Algorithm C, takes $O(n \cdot \log(n/m))$ time.

Algorithm C

```
function maxi(x, a);   {a binary search algorithm}
var low; mid; high;
begin
  low := 0; high := n div m;
  while low < high do begin
    mid := (low + high + 1) div 2;
    if scan(x, a, mid)
    then low := mid
    else high := mid - 1
  end;
  return low
end.
```

Another possible solution would be to linearly scan through x while maintaining a suitable auxillary data structure to keep track of some relevant information. These kinds of methods have a typical O(n log n) running time, and more efficient ones are not easy to find.

4.3. The main algorithm: Final solution

In what follows, we shall describe a succinct algorithm to compute maxi(x, a), which is based on a suitable variation of the halving method and has O(n + m) running time.

We first define a function half(x) which returns a subsequence of x whose length is approximately half of the length of x. The function half(x) is defined as follows. Consider a symbol σ. Let the positions of x in which σ appears be $j_1, j_2, j_3, \ldots$ in increasing order. Discard the positions j_k where k is even, and consider positions $j_1, j_3, j_5, \ldots$ only. The sequence half(x) is obtained by the above odd-selection process on each symbol that appears in x.

4.2. Example. Let

x = 01200021120222011000 1.

The symbols in the above odd-selection process are underlined. That is,

half(x) = 012012022100.

In this example we see that |x| = 22 and |half(x)| = 12.

The function half(x) can be implemented to run in O(n) time. For each symbol in x we maintain a boolean tag, initially *false*. We make one pass through x. For each symbol encountered we flip its tag, and we then collect the symbol if and only if its tag is *true*. The subsequence of x consisting of the collected symbols is half(x). We also notice that $|\text{half}(x)| \leqslant \frac{1}{2}|x| + m$. The important property, which is stated in the following theorem, is that maxi(x, a) is closely approximated by 2maxi(half(x), a).

4.3. Theorem

$|\text{maxi}(x, a) - 2\text{maxi}(\text{half}(x), a)| \leqslant 1$.

Proof. Let

I = maxi(x, a) and J = maxi(half(x), a).

We need to show that $2J - 1 \leqslant I \leqslant 2J + 1$. First, we show that $I \leqslant 2J + 1$. Let subsequence a^1 of x

have positions $j_1, j_2, j_3, \ldots, j_I$ of a symbol σ in x. Then, either occurrences of σ at positions $j_1, j_3, j_5, \ldots$ or at positions $j_2, j_4, j_6, \ldots$ appear in half(x). (Note that the second case is a possibility, since some earlier occurrences of σ in x may not show up in a^I.) In either case, at least $\frac{1}{2}(I-1)$ of the occurrences of σ from a^I show up in half(x). Considering all symbols, we see that $J \geqslant \frac{1}{2}(I-1)$. In other words, $I \leqslant 2J+1$. Now, we show that $I \geqslant 2J-1$. The proof is similar. a^J is a subsequence of half(x). Consider the J occurrences of a symbol σ of a^J in half(x). Let those be in positions $j_1, j_2, \ldots, j_J$ in half(x). Suppose k_i is the position in x corresponding to the position j_i in half(x). These positions show up in x as well as some intermediate positions, that is, $k_1, k_1', k_2, k_2', \ldots, k_{J-1}, k_{J-1}', k_J$; that is, $2J-1$ positions in all. By similar reasoning about all symbols, we conclude that $I \geqslant 2J-1$. This completes the proof. □

4.4. Example. Let us assume a = 012 and

x = 022200001011112022201222 1.

Then,

half(x) = 0220011102221.

From the fact that maxi(half(x), a) = 3, we can conclude that $5 \leqslant \text{maxi}(x, a) \leqslant 7$.

Based on Theorem 4.3, we immediately obtain the following algorithm to compute maxi(x, a).

Algorithm D

```
function maxi(x, a);   {a halving algorithm}
var i;
begin
  if |x| < 3m then return the answer by a
     simpler method;
  i := maxi(half(x), a);
  if scan(x, a, 2i + 1) then return 2i + 1
  else if scan(x, a, 2i) then return 2i
  else return 2i − 1
end.
```

4.5. Theorem. *The function* maxi *based on the halving method, as shown above, solves the longest stuttering subsequence problem in* $O(n+m)$ *time.*

Proof. The correctness of Algorithm D immediately follows from Theorem 4.3 and from induction on the length of x. To analyze the running time, let $T_m(n)$ denote the running time of maxi(x, a) with $|x| = n$ and $|a| = m$. If $n < 3m$, then the algorithm clearly takes O(m) time. If $n \geqslant 3m$, then the algorithm first invokes half(x). This takes O(n) time. Then, the function maxi is called recursively with the length of the first parameter, namely |half(x)|, being at most $\frac{1}{2}n + m$. (Note that $\frac{1}{2}n + m < n$, so the recursion terminates.) Therefore, this call takes $T_m(\frac{1}{2}n + m)$ time. The remaining portion of the algorithm which includes at most two calls of the scan function takes O(n) time. We conclude that

$$T_m(n) \leqslant cm \quad \text{for } n < 3m,$$

and

$$T_m(n) \leqslant T_m(\tfrac{1}{2}n + m) + cn \quad \text{for } n \geqslant 3m,$$

for some positive constant c. By a simple induction on n we can show that

$$T_m(n) \leqslant 6c(n+m). \quad \square$$

5. Conclusion

In this paper we have shown another application of the halving paradigm. More specifically, we have shown that the longest stuttering subsequence problem can be solved in linear time. This and other applications of the halving method that have already appeared in the literature show that the method can be used in a rather wide variety of situations.

References

[1] A.V. Aho, J.E. Hopcroft and J.D. Ullman, The Design and Analysis of Computer Algorithms (Addison-Wesley, Reading, MA, 1974).

[2] D. Dolev, K. Karplus, A. Siegel, A. Strong and J.D. Ullman, Optimal wiring between rectangles, in: Proc. 13th Ann. ACM Symp. on Theory of Computing (1981) 312–317.

[3] G.N. Frederickson and D.B. Johnson, Generalized selection and ranking: Sorted matrices, SIAM J. Comput. 13 (1) (1984) 14–30.

[4] J.L. Hodges and E.L. Lehmann, Estimates of location based on rank tests, Ann. Math. Statist. 34 (1963) 598–611.

[5] D.B. Johnson and T. Mizoguchi, Selecting the kth element in $X+Y$ and $X_1+X_2+\cdots+X_m$, SIAM J. Comput. 7 (2) (1978) 147–153.

[6] A. Mirzaian, Channel routing in VLSI, in: Proc. 16th Ann. ACM Symp. on Theory of Computing (1984) 101–107.

[7] A. Mirzaian, River routing in VLSI, J. Comput. System Sci. 34 (1) (1987) 43–54.

[8] A. Mirzaian and E. Arjomandi, Selection in X+Y and matrices with sorted rows and columns, Inform. Process. Lett. 20 (1985) 13–17.

[9] M.I. Shamos, Geometry and statistics: Problems at the interface, in: J.F. Traub, ed., Algorithms and Complexity: New Directions and Recent Results (Academic Press, New York, 1976) 251–280.

[10] D.L. Shell, A high-speed sorting procedure, Comm. ACM 2 (7) (1959) 30–32.

[11] A. Siegel and D. Dolev, The separation for general single-layer wiring barriers, in: H.T. Kung, B. Sproull and G. Steele, eds., VLSI Systems and Computations (Computer Science Press, Potomac, MD, 1981) 143–152.