A Succinct Data Structure for Dynamic Queries

Pushkar Mishra
pushkarmishra@outlook.com
University of Cambridge

Kunal Kapila
kunalapila@gmail.com
Indian Institute of Technology, Kanpur

January 2015

Abstract

Segment trees are widely used to process queries like the Range Minimum Query (RMQ), the Range Sum Query (RSQ) and Stabbing Query. In this paper, we propose an isometric data structure, the "Segbit", to accomplish these tasks. Segbit is based on the principle that a range can be decomposed into smaller ranges, whose lengths are powers of 2. The operations to traverse ranges are based on binary representation of indexes. A Segbit performs queries and updates with the same time complexity as a Segment tree, but requires strictly less than $2 * N$ memory (which, unlike a segment tree, includes the space to store the original array on which Segbit is built), for an array of size N. It is shorter to code and easier to understand. The update/query routines of a Segbit are iterative and have a lower constant of complexity, thus being faster than their recursive counterparts of the Segment tree. Overall, the Segbit performs better than Segment trees.

Keywords: Segbit; Data Structure; Arithmetic Coding; Segment tree, Queries.

1 Introduction

Consider a one-dimensional array $A[1..N]$. We wish to answer a large number of dynamic Range Minimum Queries (RMQ) on arbitrary intervals of this array. By "dynamic", we mean that the intervals to be queried are specified during run-time and each query must be answered as and when it arrives. The queries are interspersed with update operations which require us to change the the element at a particular index to a new value. The RMQ can be replaced with other queries like the Range Sum Query, the Range Maximum Query, etc. Such a set of operations, where ranges are queried and individual elements are updated is called *Point-Update Range-Query.*

We also take up the dynamic "Stabbing Query" problem. Consider a set of intervals S. Each interval in S lies between the points 1 and N. For a point q, $S(q)$ is the number of intervals in S which contain the point q. The Stabbing Query problem asks for computing $S(q)$ efficiently. By "dynamic" we mean that intervals can be added to S, or can be deleted from it. Our implementation uses $O(N)$ memory. There are methods which utilize $O(s)$ memory, where s is cardinality of set S, however, we do not compare with those methods due to their complex implementations. This problem requires us to update ranges and query individual points, and hence, belongs to the category of *Range-Update Point-Query* problems. The above listed problems are of consequence to Computational Geometry, Graphs and Data Compression.

In 1977, J.L. Bentley, in his paper[1], presented a very powerful data structure known as the Segment tree, which can solve such queries in logarithmic time. The aim of this paper is to put forward a simple, comprehendible variant to the Segment tree. In practice, the Segbit is faster and uses significantly lesser memory than a segment tree. The memory and running time efficiency are more pronounced in multidimensional versions.

De Berg *et al.*, in their book, proved that a segment tree uses $O(N)$ space.[2, p. 226] It is to be noted that this $O(N)$ space is in addition to the space required to store the original array on which the segment tree has been built. Since, a segment tree is a completely balanced binary tree, thus, its height is $\log_2 N + 1$. The first level contains 1 node, the second contains 2 nodes, the third contains 4 nodes and so on. By summing the so formed geometric progression, the total number of nodes in a segment tree comes out to be $2^{\log_2 N+1} - 1$.

Simplifying the expression:

$$\begin{aligned} 2^{\log_2 N+1} - 1 &= 2.2^{\log_2 N} - 1 \\ &= 2.N - 1 \end{aligned} \tag{1}$$

Therefore, the number of nodes in a segment tree is always less than $2 * N$. An array can be used to implement a segment tree. In such an implementation, the root is stored at 0. For a node stored at i, the left child can be stored at $2 * i + 1$, and the right child can be stored at $2 * i + 2$. In practical usage, the length of the array required to store the segment tree in this a manner exceeds $2 * N$ substantially.

Here is a table which presents the minimum length of array required to store a segment tree built for N elements:

Number of Elements	**Minimum length of Array**
1000	2044
10000	32752
100000	262140
1000000	2097148
5000000	16777200
10000000	33554416

Table 1: Length of array required to store a segment tree

Another way to implement a segment tree is by using linked list to represent the tree. A maximum of $2 * N - 1$ nodes are made, each holding a value and location of its children. This implementation also doesn't guarantee optimal space requirements due to overheads associated with each node (since extra data is to be stored at each node). In practice, this approach performs slower than arrays and is tougher to code (involve bugs). The biggest drawback arises in the case of multidimensional segment trees. Using linked list for multidimensional data structures complicates the code manifolds.

The update/query operations on a segment tree are recursive in nature. In general practice, it has been observed that iteration performs substantially better than recursion. Recursive programs also involve chances of error relating to overflow of stack. Therefore, avoiding recursion can prove to advantageous when handling large number of queries/updates.

A succinct data structure[3, 4], as defined by G.J. Jacobson in 1988, is a data structure which tries to be optimal in memory usage without compromising on the time taken for operations. A Segbit is implemented using dynamically allocated arrays. By dynamically allocated arrays, we mean those arrays, whose size is set during the run-time (these arrays can't be resized once allocated). For N elements, a Segbit uses strictly less than $2 * N$ space. This space includes the space required to store the original array on which it is built. The update and query operations of a Segbit are iterative in nature and perform better than the recursive update and query operations of the segment tree. Updating or querying a segment tree requires the height of the segment tree to be traversed at least twice. A Segbit, on an average, visits $\log N$ intervals for update or query, thus performing better while handling large number of operations.

For explaining the working of the new data structure, we shall consider the following two problems.

1. The first problem involves the following two operations:
   (a) $rmq(x, y)$: Query the minimum element in the range (x, y).
   (b) $update(x, v)$: Change the value at x^{th} index to v.
2. The second problem involves the following two operations:
   (a) $update(x, y, v)$: Add v to all the elements from the x^{th} index to the y^{th} index.
   (b) $query(x)$: Query the value of the element at the x^{th} index.

In sections 3, 4, and 5, we take up problem 1 to present the algorithms. In section 6, we talk about problem 2. Codes in C++ for all operations on the Segbit are provided. The codes have been punctuated with comments so that the reader can understand them without difficulty.

2 Preliminaries

Let us first give some general definitions for clarity:

In this paper, we use the name SB to refer to a general Segbit. By $\log N$, we mean $\log_2 N$.

In the C++ codes in this paper, we use the operator $<<$. This is the left Bitshift operator. It is a binary operator and $x << y = x.2^y$. Bitshifts can be performed on all computers without much overheads.

The least significant bit in the binary representation of a number is the position of the first 1 bit from the right. The position of the first bit from the right is taken to be 0. Consider an integer x. Let the binary representation of $x = a1b$, where b consists of all zeroes. Note that $Not(b)$ will consist of all ones.
$\implies -x = Not(x) + 1.$
$\implies -x = Not(a1b) + 1$
$\implies -x = Not(a)0Not(b) + 1$
$\implies -x = Not(a)0Not(0..0) + 1$
$\implies -x = Not(a)0(1..1) + 1$
$\implies -x = Not(a)1(0..0)$
Thus $x\& - x = (0..0)1(0..0)$
The Bitwise AND (&) of x and $-x$ yields a value 2^k, where k is the position of the least significant bit.
We now provide an example to support our proof. Let $x = 12$. In binary representation, $x = 1100$ and $Not(x) = 0011$. When we add 1 to $Not(12)$, we get 0100. Taking Bitwise AND of 1100 and 0100 gives 0100, which in decimal equals 4. Thus, least significant bit of 12 is at the 2^{nd} position.

An interval refers to a contiguous segment of an array. An interval spanning from a^{th} index to the b^{th} index (both inclusive) is represented in this paper as $[a : b]$.

Throughout this paper, "storing an interval" refers to storing the required data associated with that interval. For example, in the context of Range Minimum Queries, storing an interval will mean storing the minimum element of that interval.

3 Building the New Data Structure

In a Segbit, for each index i of the input array, a certain number of intervals, starting at i, are stored. The lengths of these intervals are powers of 2. Segbits can be implemented using array of dynamic arrays. The number of intervals stored for i^{th} index, and consequently the length of the i^{th} array, is dependent on the binary representation of i. At least one interval is stored for each index. An array of arrays can be created on all computers. Modern day programming languages allow dynamic memory allocation, wherein the size of an array can be declared during run-time. If making such a structure isn't possible, then for each index, a pointer to an array of required size can be maintained. This shall require negligible amount of extra memory. In this paper, we shall use the concept of dynamic arrays.

The number of intervals stored for index i is equal to $k + 1$ where k is the position of the least significant bit in the binary representation of i. For example, take the index 5. Its binary representation is 101 and the least significant bit is at 0^{th} position. Hence, one interval will be stored for the 5^{th} index. Similarly, 2 intervals will be stored for the 6^{th} index ($6 = 110$ in binary. The least significant bit is at 1^{st} position). For 8^{th} index, 4 intervals will be stored, and so on.

For the purpose of this discussion, we use the name SB for a Segbit. Indexes for which at least one interval is stored, the first interval is stored at $SB[i][0]$. Similarly, indexes for which at least two interval are stored, the second interval is stored at $SB[i][1]$. In general, for an index for which k intervals have been stored, the k^{th} interval is stored at $SB[i][k-1]$. The length of interval stored at $SB[i][k]$ is 2^k. In other words, $SB[i][k]$ stores the interval $[i : i + 2^k - 1]$.

The 0^{th} level of the Segbit is formed by all those intervals which are stored at $SB[i][k]$, where $k = 0$ and $1 \leq i \leq N$. Similarly, the 1^{st} level is formed of all the intervals stored at $SB[i][k]$ where $k = 1$ and $1 \leq i \leq N$. In general, the k^{th} level in a Segbit contains all the intervals of length 2^k.

We present an example of a Segbit. Consider an array $A = \{3, 5, 1, 6, 2, 8, 4\}$. The Segbit SB, shown below, is built to answer Range Minimum Queries for A.

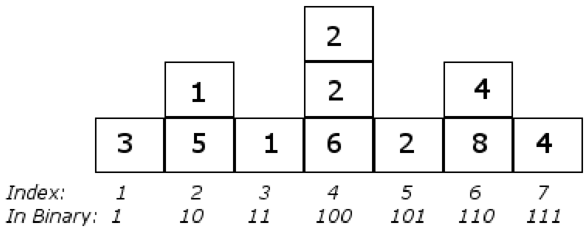

Figure 1: *Segbit for answering RMQs*

It is to be noted that the 0^{th} level in the Segbit contains the original array, i.e., $SB[i][0]$ corresponds to the original element $A[i]$. Therefore, no extra space is required for storing the original array. $SB[2][1]$ stores the minimum of the range $[2 : 3]$. $SB[4][2]$ contains the minimum of the range $[4 : 7]$, and so on.

Here is the C++ code which dynamically allocates memory and then calls the build function:

```
void main()
{
    //-------VARIABLES---------//
    int n;                      //Number of elements
    int sz=1;                   //Variable to keep a track of size to be allocated

    //------Allocating---------//
    cin >> n;
    int **SB=new int*[n+1];     //Dynamically allocating number of indexes in Segbit

    for(int i=1; i<=n; i=i*2)  //Dynamically allocating length required at each index
    {
        for(int j=1; i*j<=n; j=j+2)
```

```
            SB[i*j]=new int[sz];
        sz=sz+1;
    }
    build(SB, n);
}
```

Here is the C++ code which first takes in the input array and then builds the Segbit to store minimum values of required intervals.

```
void build(int **SB, int n){
    //-------VARIABLES----------//
    int lev=1;                  //Keeps a track of level being built
    //-------Building-----------//
    for(int i=1; i<=n; ++i)
        cin >> SB[i][0];        //Taking Input of Elements

    for(int i=2; i<=n; i=i*2){  //Building each level by divide and conquer
        for(int j=i; j<=n; j=j+i)
            SB[j][lev]=min(SB[j][lev-1], SB[j+i/2][lev-1]);
        lev=lev+1;
    }
}
```

In the code above, we use the *Divide and Conquer* paradigm. It is to be observed that $SB[i][k]$ stores the interval $[i : i+2^k-1]$. The value can be calculated in $O(1)$ time if the intervals $[i : i+2^{k-1}-1]$ and $[i + 2^{k-1} : i + 2^k - 1]$ are stored. Note that we have assumed the number of elements to be of the form $2^l - 1$, for some arbitrary l, for simplicity. For general case, it is to be ensured that while building the Segbit, we don't go beyond the number of elements.

Proposition 1. A Segbit built on N elements uses strictly less than $2 * N$ memory

Proof. For every index, at least one interval is stored. For indexes which are multiples of 2, at least 2 intervals are stored. For indexes, which are multiple of 4, at least 3 intervals are stored, and so on. Thus, we get the total memory by summing up the series $N + \frac{N}{2} + \frac{N}{4} + \cdots + \frac{N}{2^k}$, where $2^k \leq N$. Since, $2^k \leq N$, therefore the total number of terms in this series is $(\log_2 N + 1)$.

$$\begin{aligned}\text{Total Memory Used} &= \sum_{r=0}^{\log_2 N} \frac{N}{2^r} \\ &= N.\frac{1}{1-\frac{1}{2}} \\ &= 2.N\end{aligned} \tag{2}$$

The value obtained in equation (2) is a value tending to $2 * N$, and not exactly $2 * N$. The sum for an infinite geometric progression starting with one and having common ratio $\frac{1}{2}$ is 2. Here, the number of terms are finite. Thus, the memory utilized never goes beyond $2 * N$. □

From *Proposition 1*, we can say that the Segbit array can be build in time complexity O(N). This is because, there are $O(N)$ intervals to be stored and value for each interval can be calculated in constant time.

4 Range-Query Operation

In the previous section, we put forward the method to build a Segbit. In this section, we present the query routine. Consider the operation 1 of Problem 1 described in the introduction section. For the purpose of this section, we shall show how the Segbit, built in the previous section, can be used to answer Range Minimum Queries.

The algorithm for querying a range is fairly simple. Consider a Range Minimum Query for the interval $[i : j]$, on the Segbit SB, built in the previous section. To get the minimum of this range, we start at the index i of the Segbit and find the largest interval stored at i, such that the interval ends before or at j. Assume that this interval is stored at $SB[i][k]$. Thus, we have the minimum of the range $[i : i + 2^k - 1]$. We store this value in a variable $GetMin$ and move to the index $i + 2^k$. At this index, we again take the largest interval that ends before or at j. The value stored at this interval is compared with $GetMin$ and the minimum is taken. We continue this till we finally reach j. This way, Range Minimum Query is performed on $[i : j]$.

For example, let us take the case of Range Minimum Query for the interval $[3 : 13]$. We start at index 3. The largest interval that we can pick here is $[3 : 3 + 2^0 - 1]$. We store the value of this interval in $GetMin$ and move to index $3 + 2^0 = 4$. At 4, the largest interval that we can choose is $[4 : 4 + 2^2 - 1]$. We compare the value stored at $SB[4][2]$ with $GetMin$ and take the minimum and then move to index $4 + 2^2 = 8$. At 8, the largest interval stored is $[8 : 8 + 2^3 - 1]$. We can't take this and hence take the interval $[8 : 8 + 2^2 - 1]$. We compare the value here to $GetMin$ to $SB[8][2]$ and then move to index $8 + 2^2 = 12$. At 12, the suitable interval to be chosen is $[12 : 2^1 - 1]$. We compare $GetMin$ to $SB[12][1]$ and move to index $12 + 2^1 = 14$. Since, 14 is greater than our endpoint, thus we stop.

Proposition 2. The query operation takes time proportional to the logarithm of number of elements (N) for querying $[i : j]$.

Proof. We begin traversing the concerned interval at index i. Let us assume that the least significant bit of i is at k. Thus, the largest interval stored at this index spans from i to $i + 2^k$. Now, there are two cases. Either $i + 2^k \leq j$ or $i + 2^k \geq j$:

1. If the first case is true, we move to index $i + 2^k$. Let $l = i + 2^k$. The position of the least significant bit of l is at least $k + 1$. In this way, if we move forward by taking the maximum intervals at each step, we will only have to query $\log N$ intervals. This is because the maximum position of the least significant bit can be $\log N$.

2. If the second case is true, then we choose an interval of length less than 2^k. Let the chosen interval be $SB[i][m]$, where $m < k$. We move to the new index $i + 2^m$. The interval that we pick over here has to be of length strictly lesser than 2^m. This is because if we could choose an interval of length 2^m, then we would have chosen the interval $SB[i][m+1]$ rather than choosing $SB[i][m]$. Following this way, if the length keeps on decreasing, we can only move $\log N$ steps.

We have proved that an interval can be formed of a maximum of $2 * \log N$ sub-intervals. Therefore, the range queries can be done in $O(\log N)$. □

Here is the C++ code depicting the Range Minimum Query routine, for a Segbit SB, on an arbitrary interval $[i : j]$:

```
int rmq(int** SB, int i, int j){
    int mx2=C;              //Keeps a track of highest power of 2 less than
                            //distance between i and j. C>log(n)

    int lev=0;              //Keeps a track of maximum level at each index
    int GetMin=SB[i][0];    //Minimum set to the first element.

    while(i<=j){
        while(1<<mx2 > j-i+1)
            mx2=mx2-1;      //Adjusing the highest power according to current i, j
        int t=i&-i;          //Length of the largest interval stored at the index
        while(1<<lev < t)   //Adjusting maximum level according to current index
            lev=lev+1;

        int idx;
        if(mx2 < lev)        //Checking for maximum length interval possible
            idx=lev=mx2;
        else
            idx=lev;

        GetMin=min(GetMin, SB[i][idx]);
        i = i + (1<<idx);   //Moving towards j
    }
    return GetMin;          //Returning Minimum of the Range
}
```

5 Point-Update Operation

In this section, we talk about the update routine. For the purpose of this section, we shall use the Segbit SB built in section 3. Consider operation 2 of Problem 1, described in the introduction. An update operation changes the value at a particular index and recalculates the minimum for stored intervals which are concerned with that index.

When we change the value at a particular index, we need to update the data of all those intervals, which contain that index. Doing this is fairly simple. We start at the index to be updated and maintain a variable $lev = 0$, which tells us the level being updated. We update all the intervals stored at the current index and set $lev = k$, where k is the position of the least significant bit of the index. To move to the next index to be updated, we subtract 2^lev from the current index. At the new index, we update all the levels from $lev + 1$ to the maximum level, and accordingly modify lev. This procedure continues till we reach 0.

For example, consider updating index 13. We first update all the intervals stored at this index. The least significant bit for 13 is at 0^{th} position. Therefore, we move to $13 - 2^0 = 12$ and $lev = 0$. At 12, we update $SB[12][1]$ and $SB[12][2]$. Since, 2 is the position of the least significant bit of 12, thus we move to $12 - 2^2 = 8$ and $lev = 2$. At 8, we update $SB[8][3]$. We then move to $8 - 2^3 = 0$ and set $lev = 3$. We stop here.

Proposition 3. The time required to update an index is proportional to the logarithm of total number of elements.

Proof. Consider updating an index i. Let its least significant bit be at position k. This means that the binary representation of i will be of the form $a1b$, where b is a string of k zeroes and a is an arbitrary string containing zeroes and ones. Let $j = i - 2^k$. We can say that j is either 0 or of the form $c1d$, where d is a string of zeroes of at least $k + 1$ length. Assume that the least significant bit of j is at position l. Then $l \geq k + 1$. From this, we know that at least one interval of length greater then 2^k is stored at index j. Therefore, we update all the intervals stored from $SB[j][k + 1]$ to $SB[j][l]$, since they contain the index i. We then move to index $j - 2^l$ and carry out the same procedure. This can go on till we reach the maximum level in the Segbit. It is to be noted that we can visit a maximum of $\log N$ intervals because there are $\log N$ levels and at each level, only one interval contains the required index. □

Here is the C++ code depicting the update function, which changes the value of the i^{th} index to v:

```
void update(int** SB, int i, int v){
    int lev=0;              //Keeps a track of the level being updated
    while(i > 0){
        int t=i&-i;         //Length of Largest interval stored at index
        while(1<<lev <= t)  //Updating all intervals at the index
        {
            SB[i][lev] = min(SB[i][lev], v);
            lev=lev+1;
        }
        i = i - (1<<lev-1); //Moving to index containing larger intervals
    }
}
```

6 Range-Update Point-Query Operations

In this section, we take up Problem 2 described in the introduction section. The problem requires us to update all the elements in an interval by adding a constant to at each index in this interval. The problem also requires us to output the value at an index. The problem is a simpler version of the "Stabbing Query" problem.

In sections 3 and 4, we saw the operations which come under the *Point-Update Range-Query*. In order to modify the Segbit to perform *Range-Update Point-Query*, we simply need to interchange the previous two functions. Whenever we need to add a constant over an interval, we follow the query algorithm as in section 3, but instead of querying values, we add the constant to the intervals which we visit while traversing the range. Whenever we need to query an index, we keep a variable $RET = 0$. We then visit all the intervals which contain the index to be queried, using the algorithm described in section 4. The values at the visited intervals are added to RET. The final value so obtained is returned. These modifications work for clear reasons and we don't provide a proof of their correctness.

Below is the C++ code which adds a constant v to the indexes in the interval $[i : j]$. We have mentioned the modification that needs to be done. All variables and operations possess the same meaning as in the code in section 3:

```
void update(int** SB, int i, int j, int v){
    int mx2=C;
    int lev=0;

    while(i<=j){
        while(1<<mx2 > j-i+1)
            mx2=mx2-1;

        int t=i&-i;
        while(1<<lev < t)
            lev=lev+1;

        int idx;
        if(mx2 < lev)
            idx=lev=mx2;
        else
            idx=lev;

        SB[i][idx] = SB[i][idx]+v; //Constant being added to the interval
        i = i + (1<<idx);
    }
}
```

Below is the C++ code to query the value at a particular index. All variables and operations possess the same meaning as in the code in section 4:

```
int query(int** SB, int i){
    int RET=0;                    //Value to be returned
    int lev=0;
    while(i > 0){
        int t=i&-i;
        while(1<<lev <= t)
        {
            RET = RET + SB[i][lev]  //Adding all the updates done on intervals
            lev=lev+1;              //which contain the index i
        }
        i = i - (1<<lev-1);
    }
    return RET;
}
```

7 Running Time Comparison

In this section, we provide the experimental running times for RMQs and Point-Updates performed using a Segbit and a segment tree. For "Stabbing Queries" problem, similar trends were observed. The first column in the table below gives the number of elements (N) on which the Segbit and the segment tree were built. The second column gives the number of operations (Q) to be performed, which include both RMQ and Point-Update operations arranged in random order. The third and the fourth column give the time taken (in seconds) by the segment tree (T_{ST}) and the Segbit (T_{SB}) respectively, to perform all the Q operations:

N	Q	T_{ST}	T_{SB}
1023	10000	0.102	0.065
16383	50000	0.441	0.335
32767	100000	0.831	0.657
65535	500000	3.793	2.578
131071	1000000	8.232	6.217
524287	2000000	17.142	13.651
1048575	5000000	43.096	34.600
4194303	10000000	95.481	71.232

Table 2: Comparison of Running time

The implementations were done on Ubuntu 13.10 with 8GB RAM and Intel Core i3 3.1 GHz Sandy Bridge processor. The parameters for the update and query operations, i.e., the intervals to be updated, and the points to be queried were produced using the *rand*() function in the *stdlib.h* of the *C Library*.

8 Conclusion and Further Applications

In this paper, we have presented a data structure which is isometric to the already known segment tree. It performs better in terms of running time and memory requirements. The proposed data structure can easily be extended to any number of dimensions with simple and easy to comprehend codes. Multidimensional version can be used for answering "Orthogonal Queries" and other related problems. With increase in the number of dimensions, the memory and time efficiency of the Segbit is more pronounced. We claim that a data structure, built for dealing with the problems taken up in this paper, can't use lesser amount of memory than a Segbit.

The data structure can also be utilized to perform operations of *Range-Update Range-Query* category, wherein data over a range is updated and data over a range is queried. We hypothesize that such operations can be performed in $O(\log N)$. However, we put forward no claim about the optimality.

References

[1] J.L. Bentley.
Algorithms for Klee's rectangle problems. In *Technical Report*, Carnegie-Mellon University, Pittsburgh, Penn., Department of Computer Science, 1977.

[2] Mark de Berg, Marc van Kreveld, Mark Overmars, Otfried Schwarzkopf.
Computational Geometry: algorithms and applications (2nd ed.). Springer-Verlag Berlin Heidelberg New York, ISBN 3-540-65620-0, 2000.

[3] A. Brodnik, S. Carlsson, E. D. Demaine, J. I. Munro and R. Sedgewick.
Resizable Arrays in Optimal Time and Space. In *Proceedings of Workshop on Algorithms and Data Structures*, LNCS 1663, 37-48, 1999.

[4] A. Brodnik and J. I. Munro.
Membership in Constant Time and Almost Minimum Space. In *SIAM Journal on Computing*, 28(5), 1628-1640, 1999.