Algorithmica (1988) 3: 205-221

Algorithmica

An Algorithm for Segment-Dragging and Its Implementation[1]

Bernard Chazelle[2]

Abstract. Given a collection of points in the plane, pick an arbitrary horizontal segment and move it vertically until it hits one of the points (if at all). This form of *segment-dragging* is a common operation in computer graphics and motion-planning. It can also serve as a building block for multidimensional data structures. This note describes a new approach to segment-dragging which yields a simple and efficient solution. The data structure requires $O(n)$ storage and $O(n \log n)$ preprocessing time, and each query can be answered in $O(\log n)$ time, where n is the number of points in the collection. The method is best understood as the end result of a sequence of transformations applied to a simple but inefficient starting solution.

Key Words. Multidimensional searching, Functional data structures, Computer graphics, Motion-planning.

1. Introduction. This note is devoted to the following *segment-dragging* problem. Let $P = \{p_i: (x_i, y_i) | 0 \le i < n\}$ be a collection of points in the plane, represented in a Cartesian system of coordinates (Ox, Oy). For any horizontal segment AB, with $A = (a, c)$, $B = (b, c)$, and $a < b$, we define $\text{hit}(AB) = i$, where p_i is the point of maximum ordinate, such that

$$a \le x_i \le b \quad \text{and} \quad y_i \le c. \tag{1}$$

See Figure 1. If no p_i satisfies (1) then we assume that $\text{hit}(AB) = n$. If, on the contrary, more than one point have the largest ordinate among those verifying (1), we then pick the candidate with the smallest abscissa as $\text{hit}(AB)$. With these definitions at hand the problem can be formulated quite simply: implement the function hit *effectively*, using *little* preprocessing.

Segment-dragging is a common operation in computer graphics, especially for windowing tasks [3]. It can also be useful for a window manager. As observed by Mitchell [5], segment-dragging is important in certain problems of motion-planning. Also, it is often a tool for more complex operations. For example, it can be used to test whether a query rectangular box is empty, given a collection of points in any fixed dimension.

In practice, dynamic solutions—where points can be added and removed—are often preferable to static ones. We restrict our attention to the static case, however, because the solution described in this paper falls in a certain family of data structures which can be efficiently dynamized. Unfortunately, the dynamic version is complicated and can hardly be called practical. On the contrary, our static

[1] This work was started while the author was a visiting professor at Ecole Normale Supérieure, Paris, France.

[2] Department of Computer Science, Princeton University, Princeton, NJ 08544, USA.

Received August 1, 1986; revised March 2, 1987. Communicated by C. K. Wong.

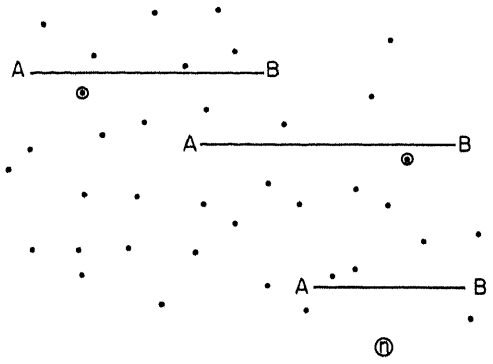

Fig. 1

solution is easy to implement and seems very efficient in practice. The size of the data structure is $O(n)$ and the time for its construction is $O(n \log n)$; moreover, any query can be answered in $O(\log n)$ time. This is a new complexity result.

Previous work on segment-dragging includes a solution in $O(n \log n)$ space and $O(\log n)$ query time [3]. As can be easily observed, segment-dragging is a special case of range searching for maximum. In the latter problem points are assigned weights, and a query requests the point of maximum weight falling in a given rectangle. Thus, to perform segment-dragging, one can use the various algorithms for range searching described in Chazelle [2]. In particular, this gives three solutions: one in $O(n)$ space and $O(\log^{1+\varepsilon} n)$ query time, for any $\varepsilon > 0$; another in $O(n \log \log n)$ space and $O(\log n \log \log n)$ query time; and a third one in $O(n \log^{\varepsilon} n)$ space and $O(\log n)$ query time, again for any $\varepsilon > 0$. A simpler, but less efficient, solution can be found in Gabow *et al* [4]. Note that the algorithm which we present here is not a substitute for these solutions since it concerns a more restrictive problem. It uses a compaction scheme introduced in Chazelle [2], as well as a number of bit-twiddling tricks, in order to produce a data structure which is both simple and efficient.

We believe that our solution is best understood as the end result of a series of transformations applied to a simple but inefficient data structure. For the sake of clarity our discussion will follow this transformational approach. In the end we will obtain a complete implementation of the algorithm in standard PASCAL. Using techniques from Chazelle [2], the algorithm can be ported to a pointer machine, that is, all address calculations can be eliminated. Admittedly, much of its simplicity is lost in the process.

2. Preliminaries. In order to confine our discussion to the essential parts of the algorithm we choose to make a number of simplifying assumptions:

(i) A query segment AB may not contribute the smallest or largest coordinate in $P \cup \{A, B\}$.

(ii) A query segment may not share coordinates with the points of P and the latter may not share coordinates among themselves.

(iii) The number of points is of the form $n = 2^w$ $(w > 0)$.
(iv) A computer word can hold any integer between 0 and $n \log n$ (we choose this value for convenience, and others, such as n, would do just as well).

It is elementary to satisfy all these conditions. For (i), the query segment can be clipped, if needed. To ensure (ii) we can go into rank space using presorting. As to (iii), if necessary, we can always pad P with extra points at "infinity" to reach the nearest power of 2. Finally, to satisfy (iv), we can use several words at once to emulate a longer computer word (if that is still not sufficient then time has come to buy yourself a new computer).

To illustrate our discussion we will use a running example throughout this paper. The points $p_0, \ldots, p_{n-1}$ will always be assumed to appear in increasing x-order. We have

$$P = \{(2, 35), (5, 17), (12, 1), (22, 13), (28, 23), (34, 3), (52, 43), (63, 15)\}.$$

3. Stage I: Getting Started. Our starting point is Bentley's *range tree* [1] applied to the set P. This gives us a complete binary tree T on n leaves: (i) the kth leaf from the left is associated with p_{k-1}; (ii) each node v is associated with the list $L(v)$ made of the points appearing at the leaves (at or) below v, sorted in increasing y-order. Figure 2 illustrates this notion. Note that we have conveniently replaced the name of each point by the value of their ordinate. There is no harm in this substitution since by assumption no two ordinates can be the same. Nodes of T have been labeled $\bar{a}, \ldots, \bar{o}$ in symmetric order. In addition, the leaves are also shown with the abscissae of the corresponding points.

Given the horizontal segment AB, with $A = (a, c)$ and $B = (b, c)$, we are now ready to *drag it down*. To begin with, we decompose AB into $O(\log n)$ so-called *canonical* pieces. To do so, we determine the leaf α (resp. β) whose corresponding abscissa immediately precedes a (resp. follows b). Let v be the nearest common ancestor of α and β. Next, we collect all the nodes $v_1, v_2, \ldots, v_p$ that are either

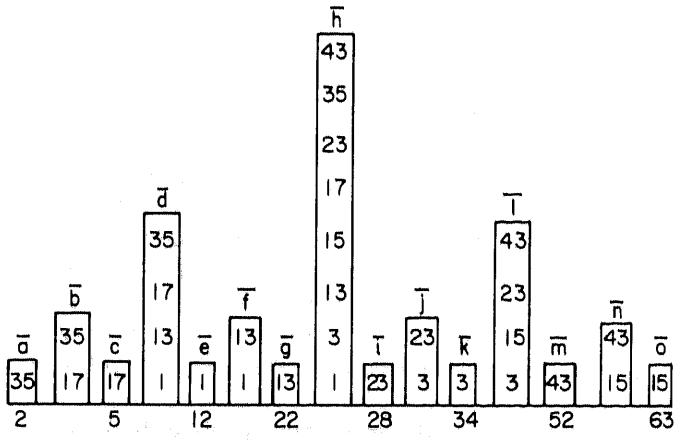

Fig. 2

right children of nodes on the path from the left son of v to α or left children of nodes on the path from the right son of v to β. Finally, for each $i = 1, \ldots, p$, we compute the entry of $L(v_i)$ whose corresponding ordinate immediately precedes the value c. Of all the points gathered in this process, we keep the one of largest ordinate; its index is precisely hit(AB). If no such point is to be found, we have hit(AB) $= n$. Turning to our running example, let us try out the method for $a = 3$, $b = 37$, and $c = 20$. We find $\alpha = \bar{a}$, $\beta = \bar{m}$, and $v = \bar{h}$. This gives $p = 3$ and

$$\{v_i \mid 1 \le i \le p\} = \{\bar{c}, \bar{f}, \bar{j}\},$$

from which we derive three candidate points: p_1, p_3, and p_5. The winner is p_1, therefore hit(AB) $= 1$.

Although we have not yet said anything about implementation, it is clear that by using binary search whenever appropriate we have here all the elements of a solution in $O(n \log n)$ space and $O(\log^2 n)$ query time. Our next task will be to bring down the storage to $O(n)$.

4. Stage II: Adding Fault-Tolerance. When faced with the task of trimming down a data structure, it is tempting to remove a fraction of the entries, and to do so in a uniform way in order to limit the damage. This will fail miserably here, however, because a single missing entry in T might entail a painful recovery operation. Looking around locally to retrieve the missing data simply will not do. This is all the more frustrating as the data structure is so evidently redundant. The remedy is to make it *fault-tolerant*. This term is not used here in the traditional sense. We say that a data structure is fault-tolerant if most of its elementary components can be reconstructed from their neighbors. For example, given two arrays of integers $A[1 \cdots n]$ and $B[1 \cdots n]$, where $B[1] = A[1]$ and $B[i] = B[i-1] + A[i]$ for $i = 2, \ldots, n$, the array B is clearly fault-tolerant; note, however, that a random permutation of its entries will likely not be so.

For our purposes here, we use a compaction scheme used previously in Chazelle [2]. The basic idea is to replace the various occurrences of a given piece of data in T by single-bit *traces*. For example, making the convention once and for all that 0 means *left* and 1 means *right*, we replace 17 by 0 in $\bar{d}$ because 17 also appears in the left child of the node (observe that it cannot appear in both children); similarly, 17 in $\bar{b}$ becomes 1 because it is found in $\bar{c}$. The transformed data structure in our running example appears in Figure 3 (ignore the numbers on the sides for the time being). Obviously, no trace need be stored at the leaves.

The ordinate of the query AB being 20, its *location* in $L(\bar{h})$ is 4: this means that $L(\bar{h})$ has precisely $(4+1)$ entries less than or equal to 20. For example, the locations of -5, 35, and 90 are respectively -1, 6, and 7. One key observation is that to find the location of 20 in $\bar{d}$ we simply have to count the number of 0's among the first five traces in the list at $\bar{h}$ and subtract 1 (the answer is 2); similarly we locate 20 in $\bar{l}$ by counting the number of 1's in the same sublist and subtracting 1, which gives 1.

Each operation can be carried out in constant time if we supplement traces with cumulative sums of 1's, as shown in Figure 3. Note that cumulative sums

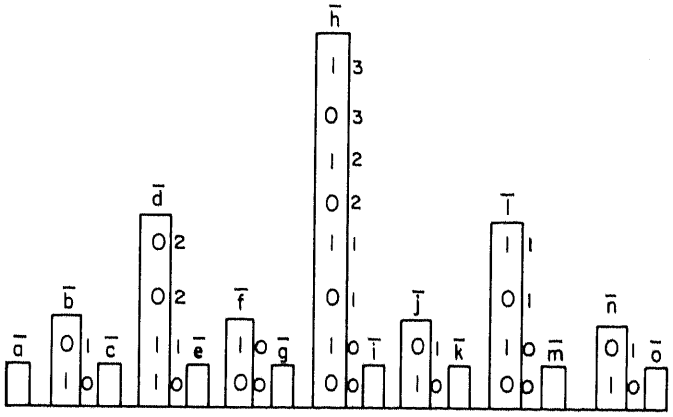

Fig. 3

of 0's are not needed because they can be readily obtained by subtracting the sum of 1's from the current location in the list.

We will now try to convince the reader that the new data structure supports segment-dragging just as well as the range tree discussed in the previous section. Of course, we still need two separate lists giving the points of P in x-order and y-order. With the latter we locate the ordinate of the query ($c = 20$) in the root $\bar{h}$ (location 4). Next, we jump to $\bar{d}$ using cumulative sums (location 2) and to $\bar{f}$ (location 1). From $\bar{d}$ we also jump to $\bar{b}$ (location 0) and then on to $\bar{c}$ (location 0). Finally, from $\bar{h}$ we jump to $\bar{l}$ (location 1) and to $\bar{j}$ (location 0). To summarize, we now have three candidate points,

$\bar{c}$: location 0,
$\bar{f}$: location 1,
$\bar{j}$: location 0,

which we must run off against each other to determine the winner. Observe that so far the running time is within $O(\log n)$.

The problem which we are now facing is to *identify* a trace, that is, reconnect it to its corresponding point (note that traces have yet to be used in our discussion). Referring to a trace as a pair (node, location), let us show how to identify, say, $(\bar{h}, 5)$. The sixth trace of $\bar{h}$ being 1 we must pursue our search with the right child of $\bar{h}$, which is $\bar{l}$. We transfer locations as usual by determining the number of 1's among the first six traces of $\bar{h}$ and subtracting 1, which yields $(\bar{l}, 2)$. The new trace now being 0, we proceed to the left and to do so we substract from the current location the number of 1's among the first three traces at $\bar{l}$, which gives the pair $(\bar{j}, 1)$. Finally, a similar operation leads to $(\bar{i}, 0)$. The identification is complete after $O(\log n)$ steps.

Returning to our running example, we identify $(\bar{c}, 0)$, $(\bar{f}, 1)$, and $(\bar{j}, 0)$ to find the names of the candidates, p_1, p_3, and p_5. The first of them, p_1, is immediately determined as the winner. The new data structure occupies $O(n \log n)$ space and supports segment-dragging in $O(\log^2 n)$ time per query. In the next section we

will see that this transformation was not as futile as it may seem right now. Despite appearances the new data structure is, for our purposes, much better than the previous one.

5. Stage III: Removing Redundancy. What we have gained by transforming T as we did is *fault-tolerance.* Indeed, a missing cumulative sum can be easily recovered by local examination. A sum can be recovered from another one, d away, by looking at d traces. This suggests trimming down the structure by keeping only one cumulative sum out of $w = \log n$ (why this choice of w will soon be obvious). In our running example, $w = 3$, so at node $\bar{h}$ we only keep locations 0, 3, and 6. Since a computer word has room for w bits (actually more, by our convention that it can assume values between 0 and $n \log n$), we can take the first w traces of $\bar{h}$ and store them as one word. This yields $2 = 010_2$, with the rest of the sequence giving $5 = 101_2$ and $1 = 01_2$. Figure 4 illustrates the reduced data structure. Before, we had nw pairs in the structure; now we only have n of them. Actually, a little more because each node may generate an extra pair if its *size* (a term from now on referring to the number of traces at the node) is not a multiple of w. At any rate, we now have a data structure of size $O(n)$ (with a rather small constant of proportionality).

Of course, the compaction makes query-answering a little more difficult. To simulate random access to a given trace, we divide its location by w to find the word which contains it. Then we truncate the word accordingly. Recall that the kth least significant bit of a number M is given by the expression

$$\lfloor M/2^{k-1} \rfloor - 2\lfloor M/2^k \rfloor.$$

The floor of an integer division is in the usual repertoire of a RAM, but exponentiation is not. This is not a problem, however; because $k \leq w$, we have $2^k \leq n$, therefore we can comfortably store the first w powers of 2 in an array of size w. This takes care of traces. To retrieve a cumulative sum, given its location, we first approximate it by reading the nearest entry stored. Then we add the missing 1's by looking directly into the word M that contains the current trace. After appropriate truncation, we finally read the desired value in an array one$[0 \cdots n-1]$, where one$[i] = \#1$'s in the binary representation of i. All these operations take constant time. To illustrate the discussion, let us determine the fifth trace of $\bar{h}$ and also jump from $(\bar{h}, 4)$ to $\bar{l}$. The desired trace appears in the kth (trace) word of $\bar{h}$ as the lth least significant bit, where $k = 1 + \lfloor \frac{4}{3} \rfloor$ and $l = 3k - 4$; its value is 0. To jump to $\bar{l}$ we compute the number of 1's among the first five traces of $\bar{h}$.

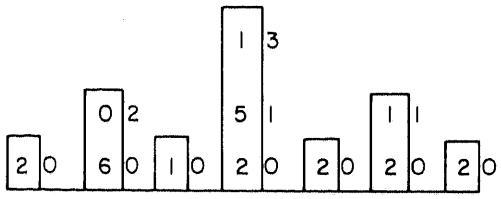

Fig. 4

As indicated above we take the kth (trace) word, 101_2, and bring the current trace (the lth least significant bit) to the lowest order position. To do so, we divide 101_2 by 2^{l-1}, which gives 10_2. Then we look up one$[10_2] = 1$, to which we add the kth cumulative sum explicitly stored (i.e., 1). Subtracting 1 from the result, we finally obtain the desired location in $\bar{l}$, that is, 1. Note that caution must be used if the current trace lies in the last (trace) word and the size of the node is not a multiple of w. As it turns out we will circumvent this difficulty altogether in the actual implementation.

To summarize, we have described a data structure of linear size which supports segment-dragging in $O(\log^2 n)$ query time. To cut down the query time we must avoid the identification of every candidate point. The idea is to have the candidates race against each other as soon as they are discovered rather than wait until they are all available. The crux is that although two traces from distinct nodes are incomparable without identification, their ordinates can be immediately compared if the traces belong to the same node.

6. Stage IV: Avoiding Repeated Identifications. For convenience, we pursue our discussion with reference to the data structure before the trimming phase. Note that for our purposes the two structures are functionally equivalent. To illustrate what follows we return to our running example. Recall that the nodes of interest (where the traces of the candidate points are to be found) are the right (resp. left) children of the nodes of the path from $\bar{d}$ (resp. $\bar{l}$) to $\bar{a}$ (resp. $\bar{m}$). The two paths play symmetrical roles, so let us restrict ourselves to the one from $\bar{d}$ to $\bar{a}$. According to the method described earlier, we locate the candidates by their traces in $\bar{f}$ and $\bar{c}$, giving locations 1 and 0, respectively. Not much can be decided on the basis of these traces alone because they belong to distinct nodes. If they were in the same list, however, things would be different. For example, the candidates have traces at $\bar{h}$ (locations 2 and 4, respectively) from which, on the basis of location alone, the second candidate appears as the winner. This suggests a two-pass approach—one, top-down, to collect traces of candidates and another, bottom-up, to run them off against each other. A more elegant method consists of keeping a current winner while going down and then avoiding the second pass altogether.

At node $\bar{d}$ the query ordinate (20) falls at location 2 (i.e., there are three points at $\bar{d}$ with a lower ordinate). Rather than jumping to node $\bar{f}$ we can simulate this process locally while staying at node $\bar{d}$. Indeed, it suffices to locate the first trace equal to 1 at or below location 2. We find location 1, which becomes our current candidate. At this point we have two locations to keep track of: the query location (value 2) and the candidate location (value 1). If these locations coincided then we could stop (why?). Since they do not, we can only infer that the winner has a trace between these two locations. In our example, the only way for the candidate to lose out later on would be for location 2 to hold the future winner (which, in fact, *is* the case). Next we jump to node $\bar{b}$ from both query and candidate locations. This takes us to locations 0 and -1, respectively. Since the trace at location 0 is precisely 1, the current candidate is updated to that location. For

two reasons at least we can now stop: the winner has been found and we are also reaching the bottom of the tree. We can now identify the candidate from its trace. (Dealing with the path from $\bar{h}$ to $\bar{m}$ in the same manner and keeping the higher point gives the final answer.)

Note that at each update of the candidate location it is important to record the new value separately, otherwise it will be lost at the next jump. This shows that in general the candidate location does *not* hold the trace of the current candidate. For example, at node $\bar{b}$ the candidate location is -1 before the update, which corresponds to no point of P whatsoever, whereas the candidate is p_3 (discovered at node $\bar{d}$).

As a matter of fact the candidate location will often hold the trace of a point with lower ordinate than the current candidate. For this reason it is imperative to update the candidate *only* when the candidate location effectively moves up. If there is a tie we apply the rule of seniority: the older candidate wins. Of course, updates of this kind must take place only at the proper *turns*: left turns from $\bar{h}$ to $\bar{a}$ and right turns from $\bar{h}$ to $\bar{m}$.

To complete our discussion we must implement a primitive which locates the next trace equal to 1 at or below a given location. One simple solution consists of providing each trace with the number of consecutive 0's right below. Naturally, by symmetry, we must also keep the number of consecutive 1's. To mix these two structures into one, we simply keep the length of the run ending at each trace. For example, at node $\bar{d}$ we store the list 1, 2, 1, 2, and at node $\bar{h}$ we keep 1, 1, 1, 1, 1, 1, 1, 1. All these lists require $O(n \log n)$ storage, but fortunately they are highly *fault-tolerant*. Thus we can sparse them out by discarding all but a fraction $1/w$. This gives 1, 2 in the case of $\bar{d}$ and 1, 1, 1 in the case of $\bar{h}$. To recover the missing information we just need to keep an array trail$[0 \cdots n-1]$, where trail$[i]$ indicates the length of the rightmost run in the binary representation of i.

Although our discussion has been kept informal and special cases have been conveniently avoided, we nevertheless have all the ingredients at our disposal for building a working algorithm. The data structure is of size $O(n)$; as we shall see shortly, it can be constructed in time $O(n \log n)$. Answering a query takes $O(\log n)$ time. In the next section we describe the preprocessing in detail. Save for minor differences it will follow precisely the outline given above.

7. The Preprocessing. We use two constants: *maxsize* specifies the maximum size of the arrays (a PASCAL requirement) and *maxint* is the largest integer used. For simplicity we declare all arrays to be of the same type.

```
type
  table = array[0..maxsize] of integer;
```

Aside from n and $w = \log n$, we also have the global variables

```
var
  n, w: integer;
  X, Y, Ysort, bit, one, tally, shift, suffix, trail: table;
```

In our running example, we have $n=8$ and $w=3$. The arrays X and Y give the coordinates of the points of P sorted in increasing x-order. For convenience, we set a sentinel $Y[n]$ to $-maxint$ (in our example $Y[n]=0$ will do); for this reason we have the condition $maxsize \geq n$.

$$X = 2, 5, 12, 22, 28, 34, 52, 63,$$

$$Y = 35, 17, 1, 13, 23, 3, 43, 15, 0,$$

$$Ysort = 1, 3, 13, 15, 17, 23, 35, 43.$$

We represent T as an implicit data structure, that is, one without pointers. Traces, for example, are all stored in a single array of nw bits. The first n bits are the traces of the parents of the leaves from left to right (level 1). The next n bits are the traces at level 2, etc. The array is broken up into pieces of length w, each fitting into a computer word (note how fortunate it is that the number of traces should be a multiple of w!). To store cumulative sums we do not use a local reference system for each node. Instead, we maintain sums of 1's with respect to the beginning of a single array. The same holds true of the list of runs. (This justifies our assumption that a computer word should be able to represent any integer up to nw; as mentioned earlier, this assumption can be easily relaxed.) What motivates this decision is the fact that the boundaries of the lists of traces do not necessarily coincide with word boundaries in the arrays. We are now ready to define the remaining variables.

(a) **bit.** For $i=0, \ldots, n-1$, the w least significant bits of bit$[i]$ are the traces from position iw to position $(i+1)w-1$. In our running example, the traces being

$$100110101100010101010101,$$

we have

$$\text{bit}[0 \cdot \cdot n-1] = 4, 6, 5, 4, 2, 5, 2, 5.$$

(b) **one.** For $i=0, \ldots, n-1$, one$[i]$ gives the number of 1's among the first $(i+1)w$ traces, that is, in the binary representation bit$[0 \cdot \cdot i]$.

$$\text{one}[0 \cdot \cdot n-1] = 1, 3, 5, 6, 7, 9, 10, 12.$$

(c) **tally.** For $i=0, \ldots, n-1$, tally$[i]$ is equal to the number of 1's in the binary representation of i.

$$\text{tally}[0 \cdot \cdot n-1] = 0, 1, 1, 2, 1, 2, 2, 3.$$

(d) **shift.** For $i=0, \ldots, w$, shift$[i]=2^i$.

$$\text{shift}[0 \cdot \cdot w] = 1, 2, 4, 8.$$

(e) **suffix.** For $i=0,\ldots,n-1$, suffix[i] indicates the length of the longest suffix in the binary representation of bit[$0\cdots i$] that constitutes a run (i.e., with all bits equal).

$$\text{suffix}[0\cdots n-1]=2,\ 1,\ 1,\ 2,\ 1,\ 1,\ 1,\ 1.$$

(f) **trail.** For $i=0,\ldots,n-1$, trail[i] is defined as suffix[i], but now with respect to the binary representation of i over w bits. Unlike the entries of suffix, note that no value in trail can exceed w.

$$\text{trail}[0\cdots n-1]=3,\ 1,\ 1,\ 2,\ 2,\ 1,\ 1,\ 3.$$

To complete this section we give some code to build the arrays above. (No attempt has been made to optimize the code.) The preprocessing consists of four procedure calls:

```
Initialize;
BitDef;
TallyDef;
TrailDef;
OneSufDef;
ShiftDef;
```

1. **Initialize.** Sets the values of n, w, X, and Y; initializes *Ysort* to Y and clears the array bit to 0.

2. **BitDef.** Computes the arrays *Ysort* and bit. The procedure is a variant of mergesort. It consists of two nested loops. The outer one takes us from one level in the merge tree to the next one above. The control variable, *level*, indicates how many leaves descend from any node at the current level. The value of *level* grows geometrically from 2 to $2^w = n$. The inner loop performs all the merges at the current level. In the running example, we have four merges at the first level, two at the second level, etc. Immediately before the merging starts, l is set to the leftmost entry to be merged and r is set to the rightmost ($r = l + level - 1$). Between merges, l increases by the value *level*. Turning now to the merging proper, the variable m indicates the rightmost location of the first list. The three variables l, r, and m are boundary delimiters; on the contrary, i and j are the running variables which control the merging (Figure 5). The array T is used as temporary storage (necessary because the sorting is not in place).

Before merging we copy the values of interest into T (setting $T[r+1]$ to *maxint* to be used as a sentinel). The details of the merging are straightforward. Concurrently, we accumulate traces in the array bit. The current location in bit is called *index*. It is updated by keeping track of a counter *fill*, which oscillates between 0 and w. To add a new trace we first shift the bits of bit[*index*] to the left to make room for the newcomer (bit[index] := 2 × bit[index]). Then, if j is incremented, we set its lower bit to 1 (bit[index] := bit[index] + 1).

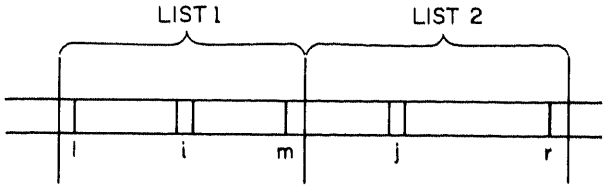

Fig. 5

```
procedure BitDef;
  var
    fill, index, level, i, j, k, l, m, r: integer;
    T: table;
begin
  fill := 0; index := 0; level := 2;
  while level <=n do
    begin
      l := 0;
      while l<n do
        begin
          r := l+level-1; m := (l+r) div 2;
          for k := l to r do T[k] := Ysort[k];
          T[r+1] := maxint; i := l; j := m+1; k := l-1;
          while (i<=m) or (j<=r) do
            begin
              bit[index] := 2* bit[index]; k := k+1;
              if (T[j]<T[i]) or (i>m) then
                begin
                  Ysort[k] := T[j]; j := j+1;
                  bit[index] := bit[index]+1
                end
              else
                begin
                  Ysort[k] := T[i]; i := i+1
                end;
              fill := fill+1;
              if fill = w then
                begin
                  index := index+1; fill := 0
                end
            end;
          l := l+level
        end;
      level := 2* level
    end
end;
```

3. **TallyDef.** Computes the array tally by enumerating the bits of each i between 0 and $n-1$.

```
procedure TallyDef;
  var
    i, j, t: integer;
begin
  for i := 0 to n-1 do
    begin
      j := i; tally[i] := 0;
      while j>0 do
        begin
          t := j; j := j div 2; tally[i] := tally[i]+t-2*j
        end
    end
end;
```

4. **TrailDef.** Computes the array trail in linear time. The idea is to find all the integers t such that trail$[t] = k-1$, for $k = 2, \ldots, w+1$. The case $k = w+1$ is taken care of separately at the beginning. For $k \leq w$, we simply enumerate all possible prefixes of length $w-k$ (variable i), to which we tack on either 1 followed by $k-1$ zeros or 0 followed by $k-1$ ones. Note that the latter number is derived from the former by subtracting 1 from it.

```
procedure TrailDef;
  var
    i, j, k, l, t: integer;
begin
  trail[0] := w; trail[n-1] := w; l := 2;
  for k := 2 to w do
    begin
      l := 2*l; j := n div l-1;
      for i := 0 to j do
        begin
          t := i*l+1 div 2; trail[t] := k-1; trail[t-1] := k-1
        end
    end
end;
```

5. **OneSufDef.** Computes the arrays one and suffix, using the arrays tally and trail, respectively.

```
procedure OneSufDef;
  var
    i: integer;
begin
  one[0] := tally[bit[0]]; suffix[0] := trail[bit[0]];
  for i := 1 to n - 1 do
    begin
      one[i] := one[i - 1] + tally[bit[i]]; suffix[i] := trail[bit[i]];
      if (suffix[i] = w) and not odd(bit[i] + bit[i - 1]) then
        suffix[i] := suffix[i - 1] + w
    end
end;
```

6. **ShiftDef.** Computes the array shift.

```
procedure ShiftDef;
  var
    i: integer;
begin
  shift[0] := 1;
  for i := 1 to w do shift[i] := 2* shift[i - 1]
end;
```

8. Answering a Segment-Dragging Query. We proceed bottom-up, starting with a few utilities which will be handy later. The function *weight* computes cumulative sums. Given a positive integer p, it returns the number of 1's among the first p traces in bit. Let $b_0, \ldots, b_{wn-1}$ be the traces in bit[$0 \cdots n-1$]: we have $weight(p) = \sum_{0 \le i < p} b_i$. In our running example, we have $weight(p) = 3, 4, 9$ for $p = 5, 7, 18$, respectively. The implementation is straightforward: locate the word of bit that contains b_p and truncate it accordingly to get the answer via tally and possibly one.

```
function weight (p: integer): integer;
  var
    i, j, z: integer;
begin
  i := p div w; j := bit[i] div shift[(i + 1)*w - p]; z := tally[j];
  if i > 0 then z := z + one[i - 1];
  weight := z
end;
```

Similarly, given a positive integer p and an integer $b \in \{0, 1\}$, $run(b, p)$ returns the length of the maximal run of the form $b_{p-k}, \ldots, b_{p-1}, b_p$, where $b_i = b$ ($p - k \le i \le p$). In our example we have $run(1, 12) = 0$ and $run(0, 12) = 3$. Note that we always have $run(0, p) \times run(1, p) = 0$. The implementation is similar to *weight*, only slightly more complicated. The only subtle point is that trail[j] may fail to give the desired answer when b and j are both 0. Indeed, in that case, trail[j] = w,

so we lose the fact that j is a truncated word. Of course, this problem does not have a symmetric instantiation for 1 (why?)

```
function run (b, p: integer): integer;
  var
    i, j, z: integer;
begin
  z := 0; i := p div w; j := bit[i] div shift[(i+1)*w-p-1];
  if j-2*(j div 2) = b then z := trail[j];
  if (j=0) and (b=0) then z := p-i*w+1;
  if (z=p-i*w+1) and (i>0) then
    if bit[i-1]-2*(bit[i-1] div 2) = b then z := suffix[i-1]+z;
  run := z
end;
```

Next, we attack the problem of jumping from the trace list of a node to the trace list of one of its children. Before going any further we must address an important question: how do we specify a node? It is tempting to say that this is not really an issue because we can directly derive the boundaries of a trace list in bit given any location in it. This is true, and will actually be used for identification purposes. However, it is inadequate for jumping. Why? Hint: think of what happens if the location is the highest in the trace list of a node distinct from the root. To deal with this problem we characterize a node by specifying explicitly the location of its first trace. For example, the root is assigned the value $(w-1)n$, the leftmost leaf is given the value $-n$, and its parent is assigned the value 0. Note that the left and right children of node $k \geq 0$ are assigned the values $k-n$ and $k-n+c/2$, respectively, where c denotes the size of node k. We easily derive that $c = 2^{\lfloor k/n \rfloor + 1}$.

We are now ready to implement the function *trans* which, given a branching direction—*branch* = 0 (left) or 1 (right)—a *node*, and a location p, jumps to the child (left or right, depending on *branch*. Recall that, by definition, a number y has location p at *node* if, within the trace list of *node*, b_p corresponds to the point of maximum ordinate less than or equal to y.

```
function trans (branch, node, p: integer): integer;
  var
    size: integer;
begin
  size := shift[node div n];
  if branch = 1 then
    trans := node-n+weight(p+1)-weight(node)-1+size
  else
    trans := p-n+weight(node)-weight(p+1)
end;
```

Given a location p in a trace list the function $id(p)$ returns the index of the point corresponding to the trace at location p. To execute this function, first we

determine the value of the trace variable (b), and then we call the function *trans* as often as needed. As soon as the current location becomes negative, we add n and we are done.

```
function id (p: integer): integer;
  var
    b, j, size: integer;
begin
  repeat
    size := 2*shift[p div n]; b := p div w;
    j := bit[b] div shift[(b+1)*w-p-1]; b := j-2*(j div 2);
    p := trans(b, (p div size)*size, p)
  until p<0;
  id := p+n
end;
```

Next on our list we have a function *path* which, given an array $A[0 \cdots n-1]$ of integers sorted in increasing order and an integer q, returns the largest index i such that $q \geq A[i]$. If no such index exists, $path(A, q) = 0$. The algorithm uses a simple binary search. (The entry $A[n]$, although accessible, is irrelevant.)

```
function path (A: table; q: integer): integer;
  var
    k, l, r: integer;
begin
  l := 0; r := n-1;
  while l<r do
    begin
      k := 1+(l+r) div 2;
      if q<A[k] then r := k-1 else l := k
    end;
  path := l
end;
```

The function *path* will serve two purposes. On the one hand it will allow us to search for the query ordinate in *Ysort*. On the other hand it will indicate the sequence of turns in T which must be taken in order to reach the nodes α and β and thus decompose the query segment.

Recall that v is the nearest common ancestor of α and β. The contest between candidates proceeds from v to α and from v to β, successively. A flag *dir* is set to 0 in the first case and 1 in the second case. What arguments should be passed to a function *cand* so that it can start off the contest? Aside from *dir* and the initial location p at the root, we also need the sequence of turns from the root to α (if $dir = 0$) or β (if $dir = 1$). This is encoded in the bits of an integer $path < n$.

Finally, we also need v; actually its size, denoted *break*, will do just as well. The function uses a few local variables: *best* holds the location of the current candidate. This is not always the same as the candidate location, denoted *try* (go back to Section 6 if this subtle difference no longer makes sense). Initially, *try* is to set to the value $(w-1)n-1$ (one slot below the first trace of the root) and *best* is set to -1. This negative value is handy to check if in the end a valid candidate has been found. A variable *size* keeps track of the current node size. It is used to determine when the race should start ($size < break$). When the branching is appropriate we test whether *best* should be updated ($try < tmp = p - run(dir, p)$). Note that *tmp* may not always correspond to a valid candidate. At worst, however, *try* is only one slot below the bottom of the current trace list. Therefore, if we have $tmp > try$, then *try* will automatically correspond to a well-defined candidate point. To conclude each loop we jump from p and *try* to the corresponding locations in the appropriate child. This child is derived from *path* and *size*.

```
function cand (dir, p, path, break: integer): integer;
  var
    b, best, node, size, tmp, try: integer;
begin
  best := -1; node := (w-1)*n; try := node-1; size := n;
  repeat
    b := (2*path) div size-2*(path div size);
    if (size < break) and (b = dir) then
      begin
        tmp := p-run(dir, p);
        if try < tmp then
          begin
            try := tmp;
            best := try
          end
      end;
    p := trans(b, node, p); try := trans(b, node, try);
    size := size div 2; node := node-n+b*size
  until (size = 1) or (try = p);
  if best >= 0 then cand := id(best) else cand := n
end;
```

Finally, the function *hit* can be implemented. Its parameters are respectively the left abscissa, the right abscissa, and the ordinate of the query segment. At the outset, the function *path* is invoked three times to find (i) the initial query location (p), (ii) the left path (*left*), and (iii) the right path (*right*). Next, a *while* loop is used to determine the size (*break*) of the nearest common ancestor v, and candidates are determined via two calls on *cand*. The answer readily follows.

```
function hit (a, b, c: integer): integer;
  var
    p, i, j, left, right, break: integer;
begin
  p := path(Ysort, c) + (w - 1)*n; break := n;
  left := path(X, a); right := path(X, b) + 1;
  while (2* left) div break = (2* right) div break do
    break := break div 2;
  i := cand(0, p, left, break);
  j := cand(1, p, right, break);
  if Y[i] < Y[j] then hit := j else hit := i
end;
```

9. Conclusions. We have completely described a new algorithm for dragging a segment amidst a set of points in the plane. The data structure requires linear storage and can be used to answer queries in logarithmic time. Experiments have shown that the algorithm performs very well in practice.

The purpose of this paper has been twofold: to present a new complexity result, and to show that the underlying algorithm can be implemented with little effort. We also hope that the transformational approach taken in explaining the data structure has been helpful to the reader. We close with an interesting open problem: designing a dynamic version of the data structure which is both simple and efficient. At present, we are able to achieve both criteria, but not simultaneously.

References

[1] J. L. Bentley, Multidimensional divide and conquer, *Comm. ACM*, **23** (1980), 214–229.

[2] B. Chazelle, A functional approach to data structures and its use in multidimensional searching, *SIAM J. Comput.* (1987) (in press). Preliminary version in the *Proceedings of the 26th Annual IEEE Symposium on Foundations of Computer Science*, 1985, pp. 165–174.

[3] H. Edelsbrunner, M. H. Overmars, and R. Seidel, Some methods of computational geometry applied to computer graphics, *Comput. Vision Graphics Image Process.*, **28** (1984), 92–108.

[4] H. N. Gabow, J. L. Bentley and R. E. Tarjan, Scaling and related techniques for geometry problems, *Proceedings of the 16th Annual ACM Symposium on Theory of Computing*, 1984, pp. 135–143.

[5] J. S. B. Mitchell, Private communication, 1986.