A Smooth Reshuffling Algorithm for Solving the Bulterman's Problem

M. C. ER*

Department of Computer Science, The University of Western Australia, Nedlands, WA 6009, Australia

This paper presents an efficient algorithm for solving the Bulterman's reshuffling problem in the spirit of smoothsort. The Bulterman's reshuffling problem is concerned with permuting an array of red and blue elements such that red elements are moved to specified positions without disturbing the relative order of blue elements. This smooth reshuffling algorithm solves the problem by utilising two processes, left sweep and right sweep; the latter is callable from the former. This algorithm is shown to be superior to a previously published algorithm which uses a brute force approach.

Received February 1987, revised March 1987

1. INTRODUCTION

The Bulterman's reshuffling problem[2] may be stated simply as follows. Given an array $x[0..N-1]$ consisting of R red elements and $(N-R)$ blue elements in random order, where $N = R*M$, $R \geqslant 1$ and $M \geqslant 1$, the object is to permute the array of elements such that (1) the relative order of the blue elements is preserved; and (2) the red elements are moved to the positions $r*M$, where $0 \leqslant r < R$. The relative order of the red elements is immaterial.

In his paper,[2] Feijen stated this problem and presented a solution to it. His solution, however, utilises a brute force approach, and is not very efficient. More specifically, his solution moves all red elements to the left end of the array without disturbing the relative order of the blue elements; then a train of red elements is rolling rightward, uncoupling the last red elements at the appropriate positions $r*M$, where $0 \leqslant r < R$. This simple brute force approach will solve the problem effectively, but does not take into account those red and blue elements that are already in place. More seriously, if all red elements already occupy the desired positions $r*M$, for $0 \leqslant r < R$, to begin with, Feijen's algorithm still collects them to the left end of the array and then restores them to their original positions. Such a mindless approach is clearly undesirable.

A similar problem also exists in *heapsort*. Specifically, if a given array is already in sorted order to begin with, *heapsort* destroys the sorted order by building a tree, and then sifts through the elements to restore them to their original positions. This problem was solved by Dijkstra by deriving a new *heapsort*, called *smoothsort*.[1] Since *smoothsort* does not disturb array elements that are already in sorted order, its best-case complexity is linear.

It is the purpose of this paper to present an efficient algorithm for solving the Bulterman's reshuffling problem in the spirit of *smoothsort*. Namely, if all red elements already occupy the correct locations to start with, one pass through the array should suffice to determine the case. In doing so, the worst-case complexity should not be increased.

* Now at Department of Maths and Computing Sciences, St Francis Xavier University, Antigonish, Nova Scotia, Canada B2G 1CO.

2. A NEW RESHUFFLING STRATEGY

To avoid disturbing red elements that are in place, obviously all red elements cannot be swept to one side of the array. Those that are in place should be skipped over while scanning the array. The actual situations, however, can be very complex – some right elements may occupy the appropriate positions, but the density of red elements may not be evenly distributed throughout the array. It is not obvious how to move things around without engaging in complex book-keeping operations.

A simple approach is to consider each red element in turn, and move it to the appropriate position. The necessity of maintaining the relative order of the blue elements means that a train of blue elements has to be moved for each insertion of a red element at the right place. Such an approach can easily lead to a quadratic algorithm.

A better approach is to move a train of red elements, as the last element of the train can swap with the blue element blocking the train, thus resulting that the blue element jumps over the train without disturbing the relative order of the blue elements. Because now each blue element moves over a larger distance, the resulting algorithm is more efficient. During the first pass of the algorithm we selectively sweep red elements leftward, leaving movable red elements in the appropriate positions, whereas those red elements that have overshot their respective positions initially should be left intact. During the second pass, the sweep is reversed rightward, moving astray red elements to appropriate positions, with a train of red elements formed whenever possible.

It turns out that the second pass can be avoided if all red elements are already in place or can be moved to the appropriate positions during the first pass. A well-known trick is to introduce a Boolean flag and set it as soon as an astray red element is found to be overshooting its appropriate position. However, once the flag is set the second pass is unavoidable. In a random distribution situation, some red elements are bound to be moved to their appropriate positions in the first pass; hence, there is no need to revisit those array segments that contain no astray red elements at all. The separation of left sweep and right sweep into two passes implies that those array segments containing astray red elements that require attention in the second pass be memorised during the first pass. This is clearly not desirable. A better approach is to conduct the (partial) right sweep during the first pass as soon as an array segment that requires attention is identified. To avoid turning the right sweep into a quadratic operation, such an array segment should be made as big as possible, but bounded by two nearest red elements that are in place or the ends of array. Hence the right sweep is carried out only when necessary.

3. THE SMOOTH RESHUFFLING ALGORITHM

A detailed implementation of the algorithm as described in the previous section turns out to be a non-trivial one. Before presenting the detailed implementation, a description of the conventions used is in order.

Two pointers, i and j, are used to delimit a left-bound train, such that i is pointing to a position one place ahead of the train, and j is pointing to the last element of the train. Thus it is obvious that $i = j$ when the train is empty. A similar convention is used for a right-bound train using p and q pointers.

Furthermore, we adopt a convention that the last element of a train, if not empty, is always a movable red element which is ready for swapping with the first blocking blue element. Such a convention indeed simplifies the algorithm, as we shall see below.

At any moment, i and j may point to a red or a blue element. Thus there are four possible combinations to be considered.

(i) $x[i]$ = red and $x[j]$ = red. This case may occur when an empty train runs into a red element, or a red train is in the process of expansion. If it is the latter, we simply advance i by one position without altering j. However, if it is the former, we have to decide whether or not this red element is a movable unit, and if it is not, whether or not it is the left boundary of an array segment that requires right sweep. To help with the determination, a variable c is introduced, which records the number of red elements to the right of j. A decision can easily be made by comparing the value of c with the number of slots supposed to be occupied by red elements. An adjustment to j and the right sweep are then carried out accordingly.

(ii) $x[i]$ = red and $x[j]$ = blue. This case should never exist given the adopted convention for representing a train. Therefore it is unnecessary to consider it further.

(iii) $x[i]$ = blue and $x[j]$ = blue. This is the trivial case as the train is empty. All we have to do is to advance the train one position leftward.

(iv) $x[i]$ = blue and $x[j]$ = red. Since the adopted convention suggests that the last red element of the train is movable, we simply swap $x[i]$ with $x[j]$ in order to move the train leftward without disturbing the relative order of blue elements. To maintain the same adopted convention, we need to test whether or not the new last red element (it *must* be red) is movable, and again if it is not, whether or not it is the left boundary of an array segment that requires right sweep. Again, an adjustment to j and the right sweep are then carried out accordingly.

An algorithm for sweeping leftward, with the above cases implemented, is shown in Fig. 1. Note that the procedure *ceiling* computes the following:

$$ceiling\ (a, b) = \left\lceil \frac{a}{b} \right\rceil.$$

```
procedure SweepLeft;
var i, j, c, k: integer;
begin
  i := N-1;
  j := N-1;
  c := 0;
  k := N;
  while i >= 0 do begin
    if x[i] = red then begin
      if (i = j) and (c < R-ceiling (j, M)) then begin
        {x[i] = red and x[j] = red}
        c := c+1;
        if (j mod M = 0) and (c = R-ceiling (j, M))
            then SweepRight (j, k);
        j := j-1;
        end;
      i := i-1;
      end
    else if x[j] = blue then begin
      {x[i] = blue and x[j] = blue}
      i := i-1;
      j := j-1;
      end
    else begin {x[i] = blue and x[j] = red}
      Swap (x[i], x[j]);
      i := i-1;
      j := j-1;
      if j mod M = 0 then begin
        {x[j] = red}
        c := c+1;
        if  c = R-ceiling  (j, M)  then
            SweepRight (j, k);
        j := j-1;
        end;
      end;
    end;
end {SweepLeft};
```

Figure 1. An algorithm for sweeping leftward.

The reversed process for sweeping rightward is similar to left sweep, with the reversal of direction. Again, there are four possible combinations of red and blue, but only three of them are valid. Here we adopt the same convention for representing a train – p points to the location one place ahead of the train, and q points to the last red element of the train. Since right sweep does not cover the whole array, but only a small segment of the array, we introduce a variable k to denote the right boundary of the array segment under consideration. The left boundary is, of course, delimited by j. The right boundary needs to be adjusted once the array segment concerned is swept over; it is simply set to the current left boundary. The detailed operation of right sweep, taking into account the three valid combinations of red and blue, is presented in Fig. 2.

A trace of the reshuffling of R red and $(N-R)$ blue elements in $x[0..N-1]$ carried out by the smooth reshuffling algorithm is shown in Fig. 3, where $N = 21$, $R = 7$ and $M = 3$.

4. AN ANALYSIS OF THE ALGORITHM

In the best-case situation where all red elements are in place, it is obvious that our reshuffling algorithm executes

```
procedure SweepRight (j: integer; var k: integer);
var p, q: integer;
begin
  if k > (j+M) then begin
    p:=j+1;
    q:=j+1;
    while p <= (k-M) do
      if x[p] = red then
        p:=p+1
      else if x[q] = blue then begin
        {x[p] = blue and x[q] = blue}
        p: =p+1;
        q:= q+1;
        end
      else begin {x[p] = blue and x[q] = red}
        Swap (x[p], x[q]);
        p:=p+1;
        q:=q+1;
        if q mod M = 0 then
          q:= q+1;
        end;
    end;
  k:=j;
end {SweepRight};
```

Figure 2. An algorithm for sweeping rightward.

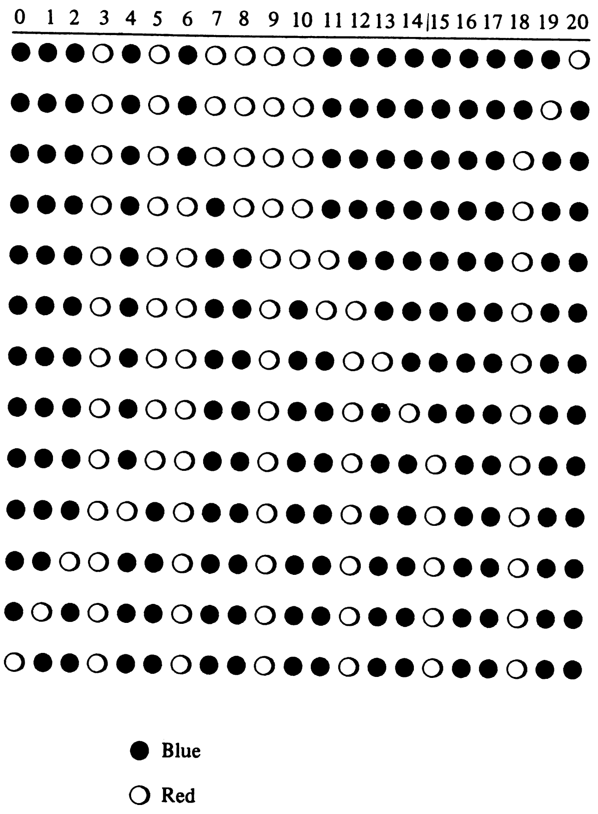

Figure 3. A trace of the reshuffling of the blue and red elements carried out by the smooth reshuffling algorithm, here $R = 7$ and $M = 3$.

the first pass through the array by using the procedure *SweepLeft* (see Fig. 1) only; the right sweep is not carried out at all. More importantly, none of the elements needs to be moved.

In the worst-case situation where all red elements overshoot their desired locations, with the exception of the leftmost one, the left sweep and the right sweep cover the whole array. Thus two full passes through the array will suffice to settle all red elements into the appropriate locations. As will be proved later, each red element moves along one direction only, either left or right, and each blue element moves only once.

In the average-case situation where red elements are randomly distributed in the array, some array segments may or may not require right sweep during sweeping leftward. Thus at least one pass through the array is required, but at most two passes through the array suffice; generally it is somewhere in between. As we shall prove below, each red element moves, if it moves at all, along one direction (left or right) at most, and each blue element moves at most once.

We summarise our claims in the following theorems.

Theorem 1

The reshuffling algorithm moves each red element along one direction at most, either left or right, but not both.

Proof. The result is trivially true if red elements are in place.

In the general case, a red element either stays to the right of the desired location or overshoots the desired location. In the former case, it will be moved by *SweepLeft* to the desired location. In the latter case, it will not be moved by *SweepLeft* at all, but will be moved by *SweepRight* to the appropriate place. Hence the result is correct. ■

Theorem 2

The reshuffling algorithm moves each blue element at most once, either leftward or rightward.

Proof. Consider a blue element. It splits the array into two halves. In the right half of the array, if the number of red elements is equal to the number of slots for red elements, the blue element concerned need not be moved at all during the reshuffling process, as this condition implies that the same equilibrium condition also exists in the left half of the array. If one of the two halves is empty, the argument is still valid.

If, however, the number of red elements is greater than the number of available slots for red elements in the right half of the array, then a left-bound train of red elements will have to pass through the blue element concerned. Hence this blue element will be moved rightward when the left-bound train runs into it. Once this train passes through it, the equilibrium state is established in the right half (and also in the left half); hence the blue element will not be moved again.

Conversely, if the number of red elements is less than the number of available slots for red elements in the right half of the array, a left-bound train, when it runs into the blue element concerned, must be empty. Thus this blue element is not moved at all during left sweeping. However, the equilibrium state for the right half is still not reached. This implies that a right-bound train must pass through it at a subsequent stage. Once a right-bound train runs into it, it will be moved leftward. After that, the equilibrium state is established and the blue element concerned need not be moved again.

This completes the proof. ■

5. CONCLUDING REMARKS

As a conclusion, we compare our reshuffling algorithm with Feijen's algorithm on a case-by-case basis.[2]

In the best-case situation, Feijen's algorithm requires two passes through the array. Each red element, with the exception of the leftmost red element, is moved leftward and then rightward. Each blue element, with the exception of the right-end ones, is moved twice, first rightward and then leftward. In contrast, our algorithm requires only one pass through the array, and moves no element at all.

In the worst-case situation, Feijen's algorithm also requires two passes through the array. Each red element is moved leftward then rightward, whereas each blue element is moved twice, first rightward and then leftward. In contrast, our algorithm also requires two passes through the array, but each red element moves along one direction only, and each blue element moves only once.

In the average-case situation, Feijen's algorithm again requires two passes through the array. Most red elements are moved leftward then rightward, whereas most blue elements are moved twice, first rightward and then leftward. In contrast, our algorithm requires one full left sweep through the array and one half right sweep, on average, through parts of the array. Each red element is moved along one direction at most; some red elements that are already in place need not be moved at all. Also, each blue element is moved at most once; some blue elements need not be moved at all.

In summary, our reshuffling algorithm is superior to Feijen's algorithm in all cases.

The lesson learned is, perhaps, more significant. In designing an efficient algorithm, one must take advantage of partial orders that already exist in the array. A brute force approach may solve the problem, but in a less efficient way. It is precisely this challenge that makes the design and analysis of algorithms so interesting.

Acknowledgement

The author wishes to thank the referee who suggested an inclusion of a pictorial representation of the Bulterman's problem.

REFERENCES

1. E. W. Dijkstra, Smoothsort – an alternative for sorting in situ. *Science of Computer Programming* **1**, 223–233 (1982).
2. W. H. Feijen, Bulterman's reshuffling problem. *Science of Computer Programming* **1**, 145–147 (1981).