

# Detecting Duplicates Among Similar Bit Vectors

(of course, with geometric applications)

Boris Aronov<sup>1</sup> and John Iacono<sup>2</sup>

**Abstract** We show how to detect duplicates in a sequence of  $k$   $n$ -bit vectors presented as a list of single-bit changes between consecutive vectors, in  $O((n+k)\log n)$  time.

**Problem** We are given a sequence  $S = \{v_1, \dots, v_k\}$  of  $k$   $n$ -bit vectors, presented as follows: The first bit vector is all zeros and each subsequent vector  $v_i$  is obtained from the previous vector  $v_{i-1}$  by flipping a single bit in position  $b_i$ ,  $0 \leq b_i < n$ .  $S$  is represented as  $b_2, b_3, \dots, b_k$ . The problem is to detect duplicates in the sequence  $v_1, v_2, \dots, v_k$ . More formally, we seek a labeling  $S \rightarrow \{1, \dots, k\}$ ,  $v_i \mapsto c_i$ , such that  $c_i = c_j$  iff  $v_i = v_j$ .

**Solution** Without loss of generality in the remainder of this note we assume that  $n$  is a power of two. Let  $T$  be the perfectly balanced binary tree on  $n$  leaves. We number the leaves of  $T$  from 0 to  $n-1$  and associate each with a bit position. Each interior node  $x$  of  $T$  is similarly associated with a block  $B(x)$  of consecutive bit positions corresponding to the leaves of the subtree rooted at  $x$ . For a bit vector  $v_i$ , let  $v_i(x)$  be its substring in  $B(x)$ . The idea behind our data structure is simple: each node  $x$  has an associated data structure that stores implicitly the set  $\bigcup_{i=1}^k \{v_i(x)\}$ . The data structure stored at node  $x$  consists of two arrays  $D_x$  and  $F_x$  that store the following data:

- $D_x[1, \dots, d_x]$  contains the sorted set including 1 and all distinct values  $i$ ,  $1 < i \leq k$ , such that  $v_{i-1}(x) \neq v_i(x)$ .
- $F_x[1, \dots, d_x]$  contains integers in the range  $1, \dots, d_x$  with the property that  $F_x[i] = F_x[j]$  iff  $v_{D_x[i]}(x) = v_{D_x[j]}(x)$ .

<sup>1</sup>Research supported in part by NSF ITR Grant CCR-00-81964 and by a grant from US-Israel Binational Science Foundation; part of work has been carried out while visiting Max-Planck-Institut für Informatik. Department of Computer and Information Science, Polytechnic University, 5 MetroTech Center, Brooklyn, NY 11201 USA; <http://cis.poly.edu/~aronov>.

<sup>2</sup>Research supported in part by NSF grant CCF-0430849. Department of Computer and Information Science, Polytechnic University, 5 MetroTech Center, Brooklyn, NY 11201 USA; <http://john.poly.edu>.

We now complete the description of our algorithm, by explaining how to initialize  $D_z$  and  $F_z$  for all leaves  $z \in T$  and how to compute  $D_x$  and  $F_x$  from  $D_l, F_r, D_l, F_r$  for any internal node  $x$  with children  $l$  and  $r$ .  $F_{\text{root}}$  describes the desired labeling of  $S$ , since  $D_{\text{root}}$  contains all the numbers  $1, \dots, k$ .

If one stores all of the leaves  $z$  in an array in numerical order, a linear scan of the sequence  $b_2, b_3, \dots, b_k$  of bit updates allows one to initialize arrays  $D_z$  and  $F_z$ , for all  $z$ . Specifically, we store the current bit vector  $v_{i-1}$  explicitly in a bit array  $V[0, \dots, n-1]$ . Since  $b_i = j$  indicates a bit flip in position  $z = j$  (recall that bit positions, and thus leaves are identified with integers  $0, \dots, n-1$ ), we flip the value of  $V[j]$ , add  $j$  to  $D_z$ , and depending on the resulting value of  $V[j]$ , set the next entry in  $F_z$  to zero or one.

---

**Algorithm 1** The pseudocode for computing  $D_x, F_x$  from  $D_l, F_l, D_r, F_r$ .

---

```

1:  $i \leftarrow j \leftarrow k \leftarrow 1$ 
2:  $D_l[d_l + 1] \leftarrow D_r[d_r + 1] \leftarrow \infty$ 
3: repeat
4:    $P_x[k] \leftarrow (F_r(i), F_l(j), k, 0)$ 
5:   if  $D_l[i] < D_r[j]$  then
6:      $D_x[k] \leftarrow D_l[i]; i \leftarrow i + 1$ 
7:   else if  $D_l[i] = D_r[j]$  then
8:      $D_x[k] \leftarrow D_l[i]; i \leftarrow i + 1; j \leftarrow j + 1;$ 
9:   else if  $D_l[i] > D_r[j]$  then
10:     $D_x[k] \leftarrow D_r[j]; j \leftarrow j + 1;$ 
11:   end if
12:    $k \leftarrow k + 1;$ 
13: until  $i = d_l + 1$  and  $j = d_r + 1$ 
14:  $d_x \leftarrow k - 1$   $\triangleright d_x$  is the length of  $P_x$  and  $D_x$ 
15: Sort  $P_x$  lexicographically on the first two fields,
    by radix sort
16: for  $k \leftarrow 2$  to  $d_x$  do
17:   if  $P_x[k-1][1] = P_x[k][1]$  and  $P_x[k-1][2] =$ 
      $P_x[k][2]$  then
18:      $P_x[k][4] \leftarrow P_x[k-1][4]$ 
19:   else
20:      $P_x[k][4] \leftarrow P_x[k-1][4] + 1$ 
21:   end if
22: end for
23: for  $k \leftarrow 1$  to  $d_x$  do
24:    $F_x[P_x[k][3]] \leftarrow P_x[k][4]$ 
25: end for

```

---

Now, we describe, for an internal node  $x$  of  $T$  with children  $l$  and  $r$ , how to construct  $D_x, F_x$  from arrays  $D_l, D_r, F_l, F_r$ ; see Algorithm 1. The new sorted array  $D_x[1, \dots, d_x]$  is built by merging the arrays  $D_l$  and  $D_r$ , eliminating any duplicates, in time  $O(d_l + d_r) =$

$O(d_x)$ . Simultaneously, we create an auxiliary array  $P_x[0, \dots, d_x]$  that records how the merge step has proceeded; it consists of quadruples of items. Refer to lines 1–13. We clearly have

**Lemma 1.** *The array  $D_x$  as constructed is correct. The first two columns of the array  $P_x$  contain pairs of integers in the range  $1, \dots, d_x$  with the property that  $(P_x[i][1], P_x[i][2]) = (P_x[j][1], P_x[j][2])$  iff  $v_{D_x[i]}(x) = v_{D_x[j]}(x)$ . The third column just numbers the rows of  $P_x$  consecutively.*

The array  $P_x$  contains all the information we need, in a sense, but not in the right order. At this point we radix-sort  $P_x$  according to the first two fields, in two passes. As  $P_x$  is of size  $d_x$  and each of these fields is a positive integer no larger than  $d_x$ , this takes  $O(d_x)$  time. In lines 16–22, we number the rearranged lines of  $P_x$  consecutively, ignoring duplicate pairs in the first two columns. These integers will be used to fill in  $F_x$  and are guaranteed to be in the range  $1, \dots, d_x$ . This takes time  $O(d_x)$ . We finally fill the array  $F_x$  using the data in  $P_x$ , as detailed in lines 23–25. A few minutes of contemplation will convince the reader that the following lemma holds.

**Lemma 2.** *The array  $F_x$  contains integers in the range  $1, \dots, d$  with the property that  $F_x[i] = F_x[j]$  iff  $v_{D_x[i]}(x) = v_{D_x[j]}(x)$ .*

**Application** Suppose one is interested in computing an arrangement of  $n$  simple shapes (such as disks, triangles, or halfplanes) in  $\mathbb{R}^2$ . There are plenty of algorithms, including deterministic ones, that can solve this problem in  $O(n^2 \log n)$  time for a variety of shapes. But what if, in addition to the face structure of the arrangement, one is interested in labeling the faces with the bit vector indicating which of the objects each face belongs to? Clearly, an explicitly stored labeling is too expensive, requiring  $\Theta(n^3)$  bits in the

worst case. However, traversing the arrangement by an Eulerian path of the face incidence graph allows one to encode the bit vectors using single-bit flips between consecutive vectors along the path. In particular using the algorithm described above, we can detect which faces correspond to identical vectors and thus are contained in identical sets of shapes. The process takes  $O(n^2 \log n)$  time. An entirely analogous process can process an arrangement of  $n$  objects in any dimension, traversing  $k$  cells in  $O((n+k) \log n)$  time, provided adjacent cells differ only in a single containment and an adjacency structure encoding local bit differences is available.

Note that the assumption that the first bit vector in the sequence is all zeros can be dropped without changing our algorithm. Also observe that our algorithm can be used with slight modifications to detect duplicates among bit vectors coming from several sequences—one just needs to artificially concatenate the sequences together by using dummy intermediate vectors, if the number of sequences is small. This will result in an additional  $O(n \log n)$  cost per concatenation. A less brute-force approach to dealing with multiple sequences which results in a  $O(n)$  concatenation cost will be described in the full version of this paper.

A geometric application with two sequences of bit vectors was presented in [1].

## Reference

- [1] P.K. Agarwal, B. Aronov, V. Koltun, “Efficient Algorithms for Bichromatic Separability,” manuscript, 2004. Preliminary version appeared in *Proceedings 15th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2004)*, 2004, pp. 675–683.