An Efficient Implementation of the Eades, Hickey, Read Adjacent Interchange Combination Generation Algorithm

Tim Hough
Computer Science Department
U.C. San Diego
La Jolla, CA 92093

Frank Ruskey *
Department of Computer Science
University of Victoria
Victoria, B.C. V8W 2Y2

Abstract

Consider combinations of k out of n items as represented by bitstrings of length n with exactly k ones. An algorithm for generating all such combinations so that successive bitstrings differ by the interchange of a single 01 or 10 pair exists only if n is even and k is odd (except for the trivial cases where $k = n, n-1, 0, 1$). This was shown by Eades, Hickey, and Read [4] (and others) but no explicit algorithm was given. Later Carkeet and Eades [3] gave an inefficient, exponential storage implementation. Here we present an implementation of the algorithm of [4] that is constant average time, and uses linear storage.

1 Introduction

Let $\mathbf{C}(n,k)$ denote the set of bitstrings of length n with exactly k ones, and $C(n,k)$ be the number of elements in $\mathbf{C}(n,k)$. We are interested in generating, or listing, the elements of $\mathbf{C}(n,k)$ so that successively listed bitstrings differ by the interchange of a single 01 or 10 pair. Let $G(n,k)$ denote the graph whose vertex set is $\mathbf{C}(n,k)$ and where an edge connects two bitstrings if they differ by a single adjacent interchange. Our problem thus becomes one of finding a Hamilton path in $G(n,k)$. This graph has two pendant vertices, 1^k0^{n-k} and $0^{n-k}1^k$, and so the Hamilton path must begin and end at those two vertices.

*Research supported by the Natural Sciences and Engineering Research Council of Canada under grant A3379.

Three groups, working independently, have shown that $G(n,k)$ has a Hamilton path if and only if n is even and k is odd (except for the trivial cases $k = n, n-1, 0, 1$). See Eades, Hickey, and Read [4], Buck and Wiedemann [2], and Ruskey [5]. In each of the papers the proof proceeds by decomposing the graph, but a different decomposition is used in each paper. The only paper to contain an explicit algorithm is that of [5] [1], where a constant average time, linear storage algorithm is given. Carkeet and Eades [3] gave an implementation of the proof of [4], but the algorithm is inefficient and uses exponential storage.

We will show that the algorithm implicit in the proof of [4] can be implemented to use linear storage and take constant average time. This is listed as an open problem in [4]. There is nothing extraordinary about the methods that we use. They could be applied to any similar proof. In comparison with the algorithm of [5], the algorithm of this paper is shorter, and probably more efficient, but is perhaps not as conceptually simple. The algorithm, implemented in Pascal, can be obtained from the second author.

The representation of combinations used in [4] was not the bitstring itself, but rather the sequence of positions that the 1's occupy. By using the bitstrings the presentation of the proof is somewhat simplified.

2 Implementation

In order to explain our implementation it is necessary to review the proof presented in [4]. The proof can be viewed as being based on the recurrence relation given below. It proceeds by induction on n and k.

$$\binom{n}{k} = \binom{n-2}{k-2} + 2\binom{n-2}{k-1} + \binom{n-2}{k}$$

First, all $C(n-2, k-2)$ bitstrings with prefix 11 are generated, followed by those beginning 01 or 10 (of which there are $2C(n-2, k-1)$), followed by the $C(n-2, k)$ with prefix 00. The list starts with the bitstring $1^k 0^{n-k}$ and ends with the bitstring $0^{n-k}1^k$. Inductively, those beginning 11 or 00 can be generated. Those beginning 11 are listed from $111^{k-2}0^{n-k}$ to $110^{n-k}1^{k-2}$, and those beginning 00 are listed from $001^k 0^{n-k-2}$ to $000^{n-k-2}1^k$. The complicated part of the proof is in listing those bitstrings that begin 01 or 10. A special kind of tree that is used in this part of the proof is defined below.

Definition: 1 A *comb* is a tree of maximum degree three where all vertices of degree three lie along a single path which is called the *spine*. The paths that are attached to the spine are called *teeth*.

[1] The authors have recently learned that Buck and Wiedemann's original report [1] contained two efficient implementations of their combination generator. However, the algorithms are presented in a little-known language, IDAL, and are not analyzed in the report.

Let us consider the specific case of $n = 8$ and $k = 5$. There are $C(n-4, k-2) = C(4,3)$ $= 4$ bitstrings with prefix 1010, and similarly there are 4 with prefix 1001, or 0110, or 0101. Inductively, the four suffixes are 1110, 1101, 1011, and 0111. The proof of [4] denotes the list of bitstrings of $m = C(n-4, k-2)$ prepended with 10 by $p_1, p_2, \ldots, p_m$, and when prepended with 01 by $q_1, q_2, \ldots, q_m$. In our example, the p list is 101110, 101101, 101011, 100111, and the q list is 011110, 011101, 011011, 010111. Note that

$$q_1, p_1, p_2, q_2, q_3, \ldots, p_{m-1}, p_m, q_m$$

is a path in $G(n-2, k-1)$. This path is the spine of the comb. The bitstrings of $\mathbf{C}(n-2, k-1)$ that begin 00 or 11 are attached as the teeth of the comb; those that begin 00 are attached to q vertices, and those that begin 11 are attached to p vertices. The tooth attached to a q vertex is obtained by moving its leftmost 1 to the right until it encounters another 1. The tooth attached to a p vertex is obtained by moving the leftmost 0 to the right until it encounters another 0. The following table lists the vertices of the spine for our example as the leftmost column, and the bitstrings to the right are the teeth of the comb.

q_1	011110			
p_1	101110	110110	111010	111100
p_2	101101	110101	111001	
q_2	011101			
q_3	011011			
p_3	101011	110011		
p_4	100111			
q_4	010111	001111		

There are two combs for $\mathbf{C}(n, k)$ depending on whether the bitstring starts 01 or 10. In other words, we have the product graph of the comb and an edge. Let us call the comb with prefix 10 the *upper* comb and the one with prefix 01 the *lower* comb. It is a simple matter to find a Hamilton path in the two combs that starts at the upper vertex $10p_m$ and ends at the lower vertex $01q_1$. Since $p_m = 10^{n-k-1}1^{k-1}$ and $q_1 = 01^{k-1}0^{n-k-2}$ the proof will be finished. The Hamilton path in the pair of combs for our example is illustrated in Figure 1.

When viewed along the spines the path sequence starts $10p_m$, $10q_m$, $01q_m$, $01p_m$. Thereafter, the patterns $01p_i$, $10p_i$, $10q_i$, $01q_i$, and $01q_i$, $01p_i$, $10p_i$, $10q_i$ alternate as i decreases from $m-1$ to 1. Of course, the teeth have to be generated along the way as well. This finishes the proof.

In order to implement the algorithm efficiently we cannot store sublists of bitstrings as was done in [3]. Our approach is to try to write a procedure `Next` that will transform the current bitstring into its successor, and only use a linear (e.g. $O(n)$) amount of auxiliary information.

From any vertex in the pair of combs there are at most three possible moves: across the

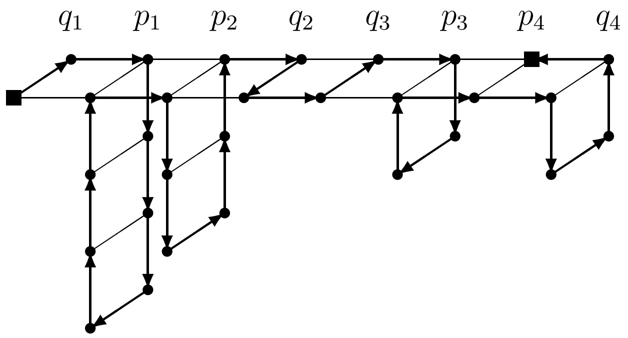

Figure 1: Path in pair of combs for $n = 8$ and $k = 5$.

comb, along the spine, or along a tooth. It is also convenient to keep track of whether we are at a p vertex or a q vertex. This leads us to the following list of 14 states.

INIT	The bitstrings beginning 11.
FINI	The bitstrings beginning 00.
PP	From a p vertex to a p vertex in same comb.
QQ	From a q vertex to a q vertex in same comb.
PQ	From a p vertex to a q vertex.
QP	From a q vertex to a p vertex.
ULP	From a p vertex in upper comb to a p vertex in lower comb.
LUP	From a p vertex in lower comb to a p vertex in upper comb.
ULQ	From a q vertex in upper comb to a q vertex in lower comb.
LUQ	From a q vertex in lower comb to a q vertex in upper comb.
TDP	Down a p tooth.
TDQ	Down a q tooth.
TUP	Up a p tooth.
TUQ	Up a q tooth.

Procedure `Next` is used recursively. In particular, when the INIT FINI, PP, and QQ moves are made, `Next` is called again to find the successor of some smaller bitstring. Procedure `Next` will have four paramenters `n`, `k`, `level`, and `dir`. Parameter `level` is the current level of the recursion, and `dir` is the direction in which the generation is proceeding. The forward direction is from 1^k0^{n-k} to $0^{n-k}1^k$ and the backward direction is the opposite. We need to be able to go in both directions because the spine is recursively traversed in the opposite direction. The basic outline of the algorithm is given below. The bitstring itself is stored in a global array `x`. When moving a single bit, the forward direction means movement to the left, and backwards means movement to the right.

```
"Initialize";
repeat Next( n, k, 1, forw );
until "x is last sequence";


procedure Next ( n, k, level, dir : integer );
begin
   if k = 1   then "move 1 one position in direction +dir" else
   if k = n-1 then "move 0 one position in direction -dir" else
   case "next move" of
     INIT: begin ... end;
     ...
     TUQ: begin ... end;
   end {case};
end {of Next};
```

The case statement contains each of the 14 states and, aside from recursive calls, we want there to be a constant amout of computation for any call to `Next`. The main complication is how to keep track of where we are in the recursive construction. This is not straightforward because the recursive steps along the spine are intermixed with non-recursive steps along the spine and up and down the teeth. This complication is overcome by introducing a global stack of records with the appropriate fields. The stack is indexed by the level of the recursion.

We have now presented the central ideas of our algorithm. The exact fields in the stack will depend on what other global information is maintained. One specific implementation will be described next. In addition to `x` and the stack there is another global array `p1` which keeps track of the positions of the 1's in the bitstring. The adjacent interchanges are done by moving a specific 1 to the right or to the left.

Each stack record contains four fields `spec`, `side`, `p`, and `nm`. Field `spec` is a boolean that keeps track of whether we are currently in the *spec*ial part of the path which is traversing the part of the graph defined by $10p_m$, $10q_m$, $01q_m$, $01p_m$. Field `side` keeps track of whether we are on the upper or lower comb. Field `p` is a counter used to keep track of which 1 is being moved when traversing a tooth. Field `nm` keeps track of which is the *n*ext *m*ove.

With the data structure described above, it is now possible to write `Next` so that a constant amount of computation is done, except for the recursive calls. Thus the running time of the algorithm is proportional to the total number of calls to `Next` that are made in generating all elements of $\mathbf{C}(n, k)$.

Let $N(n, k)$ denote the number of calls to `Next(n,k)` in generating $\mathbf{C}(n, k)$. As will be shown in the following section $N(n, k)/C(n, k)$ is proportional to n and so we do not yet have a constant average time algorithm. Upon examining the algorithm it seems that many of the calls to `Next` are wasted if we already know that the prefix is 11 or 00. By being a little more intelligent about how those cases are handled we obtain a constant average time algorithm.

The procedure Gen below (recursively) avoids the calls to Next when the prefix is 11 or 00.

```
procedure Gen ( n, k : integer );
begin
   if k = 1   then "sweep the 1 from right to left" else
   if k = n-1 then "sweep the 0 from left to right" else
   begin
      "set first two bits to be 1";
      Gen( n-2, k-2 );
      "Initialize";
      repeat Next( n, k, 1, forw );
      until "all bitstrings with prefix 01 or 10 are generated";
      "set first two bits to be 0";
      Gen( n-2, k );
   end;
end {of Gen};
```

As shown in the next section, by using Gen, we obtain a constant average time algorithm.

3 Analysis

Recall that $N(n,k)$ is the number of calls to `Next(n,k)` in generating $\mathbf{C}(n,k)$. The following recurrence relation holds: $N(n,1) = N(n,n-1) = n-1$ and otherwise

$$N(n,k) = \binom{n}{k} - 1 + N(n-2,k-2) + N(n-4,k-2) + N(n-2,k)$$

To prove the recurrence relation observe that Next is called once for everybitstring except the first, and the other terms in the recurrence follow from the recursive calls.

Let $M(n,k)$ be the number of calls to `Next(n,k)` in the modified algorithm (when Gen is used). The following recurrence relation holds: $M(n,1) = M(n,n-1) = 0$ and otherwise

$$M(n,k) = 2\binom{n-2}{k-1} - 1 + M(n-2,k-2) + N(n-4,k-2) + M(n-2,k)$$

The following Theorem shows that $M(n,k)/C(n,k)$ is indeed bounded by a constant.

THEOREM: 1

$$M(n,k) < \frac{3}{2}\binom{n}{k} \qquad \textit{and} \qquad N(n,k) < \binom{n+2}{k+1}$$

PROOF: An easy induction using the recurrence relations for N and M. ■

It would be interesting to determine M and N more exactly. From the recurrences, for fixed k, we see that $N(n,k)$ is a polynomial in n of degree $k+1$, and $M(n,k)$ is a polynomial in n of degree k. For $k=3$ we have $N(n,3) = (n-2)(n^3+2n^2+24n-120)/48$ and $M(n,3) = (n-4)(n-2)(n+3)/6$. For fixed odd $k>3$, numerical evidence indicates that

$$\frac{N(n,k)}{C(n,k)} \sim \frac{n}{2(k+1)} \qquad and \qquad \frac{M(n,k)}{C(n,k)} \sim \frac{5}{4}$$

These limits are known to be true for $k = 5,7$. If $k > 5$ then the limits are approached from above, not below.

References

[1] M. Buck and D. Wiedemann. Gray codes of combinations. Technical Report IDA-CRD Log No. 80503, Institute for Defense Analyses, 1980.

[2] M. Buck and D. Wiedemann. Gray codes with restricted density. *Discrete Math.*, 48:163–171, 1984.

[3] M. Carkeet and P. Eades. A subset generation algorithm with a very strong minimal change property. *Congressus Numerantium*, 47:139–143, 1985.

[4] P. Eades, M. Hickey, and R.C. Read. Some Hamilton paths and a minimal change algorithm. *JACM*, 31:19–29, 1984.

[5] F. Ruskey. Adjacent interchange generation of combinations. *J. Algorithms*, 9:162–180, 1988.

4 Appendix

This appendix contains a Pascal implementation of the algorithm described in the paper above. It is not intended to appear in publication. To simplify the indexing, the bitstrings are indexed from n to 1 (so that the leftmost two bits as described in the paper correspond to `x[n]` and `x[n-1]`).

```
program EHR ( input, output );

const
```

```
  MAX = 100;

  left = +1;   forw = +1;
  riht = -1;   back = -1;


type

  Moves = (TUP,TUQ,TDP,TDQ,PQ,QP,PP,QQ,
           ULP,ULQ,LUP,LUQ,INIT,FINI);

  StackElement = record
                   spec : boolean;
                   side : (lower,upper);
                   p    : integer;
                   nm   : Moves;    {The next move}
                 end;


var

  n,k   : integer;
  Count : integer;
  x     : array [1..MAX] of 0..1;      {The bitstring}
  p1    : array [1..MAX] of integer;
  stk   : array [1..MAX] of StackElement;


procedure WriteCombination;
var i : integer;
begin
  Count := Count + 1;
  write( Count:5,':' );
  for i := n downto 1 do write( x[i]:2 );
  writeln;
end {of WriteCombination};


procedure Move ( n, dir : integer );
begin
  Count := Count + 1;
  x[p1[n]] := 0;
  p1[n] := p1[n] + dir;
  x[p1[n]] := 1;
end {of Move1};
```

```
procedure Init ( dir, level : integer );
begin
  with stk[level] do begin
     if dir = forw then begin
        nm := INIT;  spec := true;  side := upper;
     end else {dir = back} begin
        nm := FINI;  spec := false;  side := lower;
     end;
     p := 1;
  end {with};
end {of Init};


procedure Next ( n, k, level, dir : integer );
begin
   with stk[level] do begin
      if k = 1 then begin
         Move( 1, -dir );
      end else
      if k = n-1 then begin
         if dir = forw then if p1[1] = 2 then p := 1;
         if dir = back then if p1[k] = k then p := k;
         Move( p, -dir );  p := p + dir;
      end else
      case nm of
       INIT:
           if dir = forw then begin
             if p1[k-2] = k-2 then begin
                Move( k-1, riht );
                spec := true;  side := upper;  nm := PQ;
             end else begin
                if p1[1] = n-k+1 then Init( forw, level+1 );
                Next( n-2, k-2, level+1, forw );
             end;
           end else {dir = back} begin
             Next( n-2, k-2, level+1, back );
           end;
       FINI:
           if dir = forw then begin
             Next( n-2, k, level+1, forw );
           end else {dir = back} begin
             if p1[1] = n-k-1 then begin
                Move( k, left );
```

```
          nm := LUQ;  spec := false;  side := lower;
       end else begin
          if p1[k] = k then Init( back, level+1 );
          Next( n-2, k, level+1, back );
       end;
     end;
 PP: begin
       Next( n-4, k-2, level+1, -dir );
       if (dir = back) and (p1[k-2] = k-2) then begin
          nm := PQ; spec := true;
       end else
         if x[n-4] = 0 then nm := LUP
         else begin nm := TDP; p := k-2; end;
     end;
 QQ: if dir = forw then begin
       if p1[1] = n-k-1 then begin
          Move( k, riht );  nm := FINI;
          stk[level+1].p := 1;
       end else begin
          Next( n-4, k-2, level+1, back );
          if x[n-4] = 1 then nm := LUQ else nm := TDQ;
       end;
     end else {dir = back} begin
       Next( n-4, k-2, level+1, forw );
       if x[n-4] = 1 then nm := LUQ else nm := TDQ;
     end;
 PQ: begin
       Move( k-1, riht );
       if x[n-4] = 1 then nm := ULQ else nm := TDQ;
     end;
 QP: begin
       Move( k-1, left );
       if (dir = forw) and spec then begin
          nm := PP; spec := false;
       end else
          if x[n-4] = 1 then begin nm := TDP; p := k-2; end
          else nm := ULP;
     end;
ULQ: begin
       Move( k, riht );
       if x[n-3] = 0 then nm := TUQ else
       if spec then nm := QP else nm := QQ;
       side := lower;
     end;
LUQ: begin
```

```
          Move( k, left );
          if x[n-3] = 0 then nm := TUQ else nm := QP;
          side := upper;
        end;
  ULP: if (dir = back) and spec then begin
          Move( k-1, left );
          nm := INIT;
        end else begin
          Move( k, riht );
          if x[n-3] = 1 then nm := TUP else nm := PP;
          side := lower;
        end;
  LUP: begin
          Move( k, left );
          if x[n-3] = 1 then nm := TUP else nm := PQ;
          side := upper;
        end;
  TUQ: begin
          Move( k-1, left );
          if x[n-3] = 1 then
             if spec then nm := QP else
             if side = upper then nm := QP else nm := QQ;
        end;
  TDQ: begin
          Move( k-1, riht );
          if p1[k-1] = p1[k-2] + 1 then
            if side = upper then nm := ULQ else nm := LUQ;
        end;
  TUP: begin
          Move( p, riht );
          if p = k-2 then begin
             if side = upper then nm := PQ else nm := PP;
          end else p := p + 1;
        end;
  TDP: begin
          Move( p, left );
          if x[p1[p]-2] = 0 then begin
             if side = upper then nm := ULP else nm := LUP;
          end else p := p - 1;
        end;
  end {case};
end {with};
end {of Next};
```

```
procedure Initialize;
var i : integer;
begin
  for i := 1 to n-k do x[i] := 0;
  for i := 1 to k do begin
     x[n-k+i] := 1;  p1[i] := n-k+i;
  end;
  Count := 0;
end {of Initialize};


procedure G ( n, k : integer );
var i : integer;
begin
  if k = 1 then begin
     for i := 1 to n-1 do Move( 1, riht );
  end else
  if k = n-1 then begin
     for i := 1 to n-1 do Move( i, riht );
  end else begin
     G( n-2, k-2 );
     Move( k-1, riht );
     with stk[1] do begin
        spec := true;  side := upper;  nm := PQ;
     end;
     Init( back, 2 );
     repeat Next( n, k, 1, forw );
     until (p1[k] = n-1) and (p1[1] = n-k-1) and (p1[k-1] = n-3);
     Move( k, riht );
     G( n-2, k );
  end;
end {of G};


begin
  write( 'Enter n and k: ' );  readln( n, k );
  Initialize;
  G( n, k );
  WriteCombination;
end.
```