September, 1982

FAST MINIMAL DISTANCE ENUMERATION OF SMALL COMBINATIONS

W.L. Steiger[1]
P.M. Neuss[2]

DCS-TR-119

[1]Department of Computer Science
Rutgers University
New Brunswick, New Jersey 08903

and

[1]Department of Statistics
Princeton University

[2]Department of Computer Science
Carnegie-Mellon University

TITLE: Fast Minimal Distance Enumeration of Small Combinations.

AUTHORS: W.L. Steiger
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

and

Department of Statistics
Princeton University
Princeton, NJ 08544

P.M. Neuss
Department of Computer Science,
Carnegie-Mellon University
Pittsburgh, PA

KEYWORDS AND PHRASES: combinations, pointers, constant weight codes, Hamiltonian circuits on the N-cube, binomial coefficients

CR CATEGORIES: 5:30

ACKNOWLEDGEMENT:

We thank Bruce Ladendorf for valuable suggestions. The first named author wishes to thank theDepartment of Statistics, Australian National University, and the Statistical Laboratory, Cambridge University, where he was Visiting Fellow in 1979-80 while this work was being carried out. Computations were done on the Rutgers Laboratory for Computer Science Research DEC20.

ABSTRACT

We give a history dependent algorithm that satisfies the claims of the title. It has other desirable attributes as well. It is computationally much simpler than algorithms studied in recent work of Payne and Ives when, in enumerating n objects k at a time, k is small compared to n. In fact $SN(n,k)$ decreases $n \to \infty$ where SN denotes the complexity of the present method and PI, that of Payne-Ives. This is probably due to the savings in overhead required by historyless enumeration.

FAST MINIMAL DISTANCE ENUMERATION OF SMALL COMBINATIONS

INTRODUCTION

Interest in orderly enumeration of combinations or equivalently, in subsets of a given set, arises from many applications. For example in fitting a k^{th} degree polynomial P to $n > k+1$ data points (x_i, y_i) by minimizing the sum $f(P) = \sum_{1}^{n} |y_i - P(x_i)|$, of absolute deviations, the fit is determined by a certain set S of $k+1$ data points through which the polynomial passes. The optimality condition for S is that no point not in S can replace a point in S to give a polynomial with a smaller value of f. However in degenerate cases, although S is optimal in the above sense, there is a set $T \supset S$ of $m > k+1$ points for which $y_i = P(x_i)$, $(x_i, y_i) \varepsilon\ T$. To terminate correctly in such cases, the curve fitting algorithms must go beyond the optimality of S; they must examine <u>all</u> the $\binom{m}{k+1}$ subsets of size $k+1$ of T, stopping only if all are optimal. (see[2])

A requirement in the foregoing example calls attention to an important general principal. In listing the different subsets of size $k+1$, and testing each for optimality, it is computationally simplest if successive subsets have k points in common. This property of the enumeration is <u>minimum distance</u>, or <u>distance two</u>, so-called because if the m points are each assigned a mark of 0 or 1, $k+1$ 1's in all, "minimum distance" means that two marks, a 0 and a 1, are exchanged in a single step of the enumeration.

Another feature of the example that is sometimes a general concern is that m and k are close, or that $m-k$ is small in comparison to m.

An enumeration algorithm suited to this application would have to be specially fast and efficient in such cases.

The literature on combinations is fairly recent. Mifsud [6] and Kurtzburg [4] give early algorithms for orderly enumeration, the first being lexicographic. However neither method was minimum distance.

Chase [3] devised TWIDDLE, a fast method with the additional property of bidirectionality - the enumerated sequence can easily be produced in reverse order. However TWIDDLE is not historyless. The next combination cannot be determined only from knowledge of the current one. While these latter two properties may be important in some contexts, they do not seem to be advantageous in the curve fitting example.

Finally Liu and Tang [5] devised a fast, historyless, distance two algorithm using Gray codes. Later, using binary reflected Gray codes, Bitner, Ehrlich and Reingold [1] gave an improved algorithm that also generated the Liu-Tang sequence. The defect of both these methods is that they are stated in terms of n marks $m_1,\ldots,m_n$ of which k are 1 and $n-k$ are 0. It is more complex to manipulate the n marks directly than to operate on k pointers, $j_1,\ldots,j_k$ describing which marks are 1's.

This was the focus of the work of Payne and Ives [7] who produced pointer versions of the foregoing algorithm and compared their computational complexities with that of several of the other methods mentioned above. The main conclusion is that in terms of total operation counts for several choices of n and k (compare, assign, arithmetic, logical) their pointer

version of Liu-Tang is superior for large n and small k (they have also rendered it bidirectional). One of the points of the present paper is that historyless enumeration pays a certain price in complexity. An algorithm simply designed to take n marks (or k pointer values) and produce the next combination must have a certain fixed overhead for each combination. A history dependent method that could produce many members of the generated sequence from the current one would spread this cost over many combinations and save time, overall.

Our algorithm, like TWIDDLE, is history dependent and distance two. We compare it to the methods mentioned in [7] and, when k is small compared to n, it is fastest. In fact, for a variety of complexity measures for SN and PI,

$$\frac{SN(n,k)}{PI(n,k)} \downarrow$$

as $n \uparrow$, SN denoting the complexity of generating the $\binom{n}{k}$ combinations by the present algorithm, PI the complexity of Payne-Ives. Finally it is an easy matter to endow the algorithm with a property similar to bidirectionality and also to give some attributes of historyless enumerations, should these be attractive features.

Note: After this paper was completed Lam and Soicher (CACM, August '82) published a promising new combination generator. We have not compared its performance to ours. However we believe that as n grows, with k fixed, the Lam-Soicher method will require about 50% more time than ours. This is because asymptotically, (1) our method performs one assignment statement per generated combination and (2) the proportion of work required by recursion decreases to zero. However it remains to actually make the comparison.

THE ALGORITHM AND ITS PROPERTIES

Given k and n, assumed to be much larger than k, the algorithm enumerates the $\binom{n}{k}$ combinations by changing k pointer values in a minimal distance fashion. For ease of exposition we add two dummy pointers $j_0 = 0$ and $j_{k+1} = n+1$. Initially, $j_i = i$, $i=1,\ldots,k$.

The algorithm is based upon two procedures, UP and DOWN. The first takes $m \leq k$ successive pointer values, starting with the i^{th} one, j_i. They are assumed to be consecutive so

(1) $$j_{\ell+1} = j_{\ell+1}, \ell = i,\ldots,i+m-2$$

If $j_{m+i-1}+1 = j_{m+i}$, the m given pointer values are consecutive with the next value, so the procedure simply outputs as the current combination. Otherwise it changes pointers $(j_i,\ldots,j_{i+m-1})$ in a minimal distance fashion until they are again consecutive and abut j_{m+i}. At each change, $J = (j_1,\ldots,j_k)$ is output.

The way in which UP enumerates $(j_i,\ldots,j_{i+m-1})$ until j_{i+m-1} abuts j_{i+m} is to

(i) enumerate $(j_{i+1},\ldots,j_{i+m-1})$ UP to j_{i+m} by minimal distance changes.

(ii) set $j_i \leftarrow j_i+1$

(iii) if not complete, enumerate $(j_{i+1},\ldots,j_{i+m-1})$ DOWN to j_i

(iv) set $j_\ell \leftarrow j_\ell+1$, $\ell = i,\ldots,i+m-1$

(v) go to (i) if not complete

Step (i) assumes $m \geq 2$. If $m = 1$, the single pointer j_i is incremented until it abuts j_{i+1} .

Also the case $j_{i+m} = j_{i+m-1} + 2$ is treated separately to avoid excess recursion. Here, the space in

(2) $\underset{j_i}{\bullet}\ \bullet\ \bullet\ \bullet\ \bullet\ \underset{j_{i+m-1}}{\bullet}\ \underset{j_{i+m}}{\bullet}$

is moved successively to the left by performing

(3) $j_{i+\ell} \leftarrow j_{i+\ell} + 1$

successively for $\ell = m-1$ down to $\ell=0$. The DOWN referred to in (iii) is a mirror image of UP, moving $(j_{i-m+1},\ldots,j_i)$ with minimal distance changes until $j_{i-m} + 1 = j_{i+1-m}$. The two procedures, coded in PASCAL, and using an output routine OUTPUTJ, are as follows:

```
PROCEDURE  UP(I,M:  INTEGER);
VAR
   E,TEMP:  INTEGER;
BEGIN
   E  := J[I+M];
   IF E-J[I+M] = 1  THEN OUTPUTJ
      ELSE IF M = 1  THEN FOR TEMP  := J[I] TO E-1 DO
         BEGIN
            J[I]  := TEMP;
            OUTPUTJ;
         END
      ELSE IF E-J[I+M-1] = 2  THEN
         BEGIN
            OUTPUTJ;
            FOR TEMP  := I+M-1  DOWNTO I DO
               BEGIN
                  J[TEMP] := J[TEMP] + 1;
                  OUTPUT
               END
         END
      ELSE BEGIN
         UP(I+1,M-1);
         J[I]  := J[I] +1;
         DOWN(I+M-1,M-1);
         FOR TEMP  :=  I TO I+M-1 DO J[TEMP] :=  J[TEMP] + 1;
         UP(I,M);
         END;
```

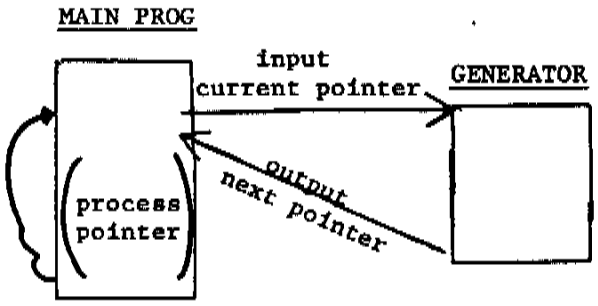
```
PROCEDURE  DOWN(I,M:  INTEGER);
VAR
   E,TEMP:  INTEGER;
BEGIN
   E  := J[I-M];
   IF J[I-M+1]-E = 1 THEN FOR TEMP  := J[I] DOWNTO E+1 DO
      BEGIN
         J[I]  := TEMP;
         OUTPUTJ;
      END
      ELSE IF J[I-M+1]-E = 2 THEN
         BEGIN
            OUTPUTJ;
            FOR TEMP  := I-M+1 TO I DO
               BEGIN
                  J[TEMP]  :=  J[TEMP]-1;
                  OUTPUTJ;
               END
            END
         ELSE BEGIN
            DOWN(I-1,M-1);
            J[I]  := J[I]-1;
            UP(I-M+1,M-1);
            FOR TEMP  :=  I DOWNTO I-n+1 DO J[TEMP]  := J[TEMP]-2;
            DOWN(I,M);
            END;
END;
```

Clearly the two procedures could be combined in one, succinct coding. However the required logic to decide whether the current phase of enumeration is UP or DOWN, and to deal with these different instances, would increase complexity and cost in terms of time.

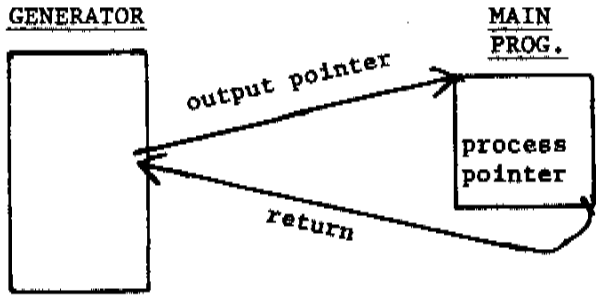
Here is the algorithm to enumerate in a forward fashion. Initialize $j_i \leftarrow i$, $i=0,\ldots,k$, $j_{k+1} \leftarrow n+1$ and UP(1,K). To get another sequence, start with $j_i \leftarrow n-k+i$, $i=1,\ldots,k+1$, $j_0 \leftarrow 0$ and DOWN(K,K) .

The algorithm can be used in enumeration with an arbitrary start, a set of pointer values, $J=(j_1,\ldots,j_k)$. Scrutiny of J shows where in the previous enumeration sequence J is, and what the stack of calls to UP and DOWN would look like. For example if j_1 is even the bottom of the stack is DOWN (K,K-1), etc. If each output in UP and DOWN is counted, terminating after $\binom{n}{k}$ steps, and if, when the stack is empty $j_i \leftarrow i$, $i=1,\ldots,k$, is performed, then UP(1,K), the procedure will be rendered cyclical. This is a useful property possessed by historyless methods, but not characteristic of them.

One advantage of historyless enumeration is that, because it puts out the next combination given the current one, it can be imbedded in a main program that does some processing on each generated combination. Thus,



In history dependent generation, once the current pointers have been processed, the generator must continue from the state in which it was suspended. This type of process is commonly called a co-routine. If the implementation language doesnt offer the co-routine capability, history dependent generation may be achieved in several other ways. First, the processing of each generated combination could be imbedded in the generation routine as in



This is undesirable because the generation then is a different program in each application. Also, the main program is subordinate to the generation routine.

Another possibility is to list the entire combination sequence in a file. The main program then reads the list successively. In this case, the time-savings of history dependent generation would probably be lost in input-output costs.

When k is small compared to n the algorithm should be very fast because (1) there is little overhead in managing the few recursive function calls and (2) much of the enumeration is based on $j_i \leftarrow j_i+1$ or $j_i \leftarrow j_i-1$. These intuitions are supported by analysis and experiment.

We compared the present algorithm with both Payne-Ives' pointer versions of Liu-Tang, with their pointer version of TWIDDLE, and with the Bitner, Ehrilich, and Reingold algorithm. It is only worth reporting the results of the faster version of Liu-Tang because the other methods are much slower.

A FORTRAN implementation of our algorithm was used. It handles recursion via stacks, thus realistically reflecting the overhead of recursion. We felt this was preferable to comparing (1) a PASCAL implementation of our algorithm to Liu-Tang in FORTRAN or (2) translating Liu-Tang into PASCAL.

The following table shows the total CPU time (on a DEC 20) for both the current method, SN, and Payne-Ives, PI, for various n and k. It seems safe to assert that SN/PI $\downarrow$ as $n \uparrow$ for any k.

In fact it is likely that as $n \to \infty$, $SN(n,k)/PI(n,k)$ converges to some function $a_k < 1$ of k. It is easy to show that with the present algorithm, for any $k \geq 1$, the number of assignment statements executed per generated combination decreases to 1 as $n \to \infty$. The analysis is more complex for the Liu-Tang algorithm although any limit point is clearly > 1.

TABLE 1

CPU times for SN and PI for various values of n and k.

	n							
	50	100	150	200	250	300	350	
SN	.12	.82	2.61	5.63	10.37	17.71	27.71	
PI	.20	1.60	5.65	12.75	23.75	41.55	66.23	k=3
SN/PI	.600	.513	.462	.442	.437	.426	.418	

	n							
	50	60	70	80	90	100	110	
SN	1.84	3.37	5.81	9.40	14.41	22.05	31.32	
PI	2.50	5.32	8.99	15.37	24.63	37.82	55.83	k=4
SN/PI	.736	.633	.646	.612	.585	.583	.561	

	n						
	20	30	40	50	55	60	
SN	.26	1.64	6.11	17.48	27.35	40.67	
PI	.19	1.55	7.07	21.12	34.86	53.44	k=5
SN/PI	1.37	1.06	.864	.828	.785	.761	

	n					
	20	25	30	35	40	
SN	.60	2.44	7.23	17.83	38.95	
PI	.46	1.97	6.58	17.67	39.77	k=6
SN/PI	1.30	1.24	1.10	1.01	.98	

TABLE 2

CPU time per 10^6 combinations for SN and PI based on times in Table 1

	n							
	50	100	150	200	250	300	350	
SN	6.12	5.07	4.73	4.29	4.03	3.98	3.91	k=3
PI	10.20	9.96	10.25	9.71	9.23	9.33	9.35	

	n							
	50	60	70	80	90	100	110	
SN	7.99	6.91	6.34	5.94	5.64	5.62	5.43	k=4
PI	10.86	10.91	9.80	9.72	9.64	9.64	9.67	

	n						
	20	30	40	50	55	60	
SN	16.77	11.51	9.29	8.25	7.86	7.45	k=5
PI	12.25	10.88	10.74	9.97	10.02	9.78	

REFERENCES

1. Bitner, J.R., Ehrlich, G., and Reingold, E.M. "Efficient generation of the binary reflected Gray code and its application". Comm ACM 19,9 (Sept. 1976), 517-521.

2. Bloomfield, P. and Steiger, W.L. "Least absolute deviations curve fitting". Siam J. Scientific and Statistical Computing 1, (June 1980), 290-301.

3. Chase, P.J. "Algorithm 382. Combinations of M out of N objects", Comm ACM 13,6 (June 1970), 368.

4. Kurtzburg, J. "Algorithm 94", Comm ACM 5,6 (June 1962), 344.

5. Liu, C.N. and Tang, D.T. "Algorithm 452. Enumerating combinations of m out of n objects", Comm ACM 16,8 (Aug. 1973), 485.

6. Mifsud, C.J. "Algorithm 154. Combination in lexicographic order", Comm ACM 6,3 (March 1963), 103

7. Payne, W.H. and Ives, F.M. "Combination generators", ACM Trans. on Math. Software 5,2 (June 1979), 163-172.