A Minimum-Change Algorithm for Generating Lattice Points

Derek O'Connor
University College, Dublin

June 25, 2006

Abstract

This paper presents a simple minimum-change algorithm to generate all lattice points in a given volume. The running time of the algorithm is $O(N)$, where N is the number of points to be generated

Keywords & Phrases. Combinatorial Generation, Minimum-Change Algorithms, Gray Codes, Computational Probability.

This is a tidied-up version of a paper written in the mid-1980s. This problem arose in an effort to speed up an enumerative algorithm for solving the Stochastic PERT Problem. See `http://www.derekroconnor.net/PERT/Pert2006.pdf`

1. Introduction

Combinatorial problems often require the systematic or random generation of some or all combinatorial objects in a given class, e.g., partitions of a set, subsets of a set, permutations of a set of integers.

Optimization problems such as

$$\min\ f(x_1, x_2, \ldots, x_n),$$
$$\text{subject to :}\quad \text{all } n\text{-tuples } (x_1, x_2, \ldots, x_n) \quad \text{satisfy certain constraints},$$

may require the generation of all n-tuples that satisfy the constraints and the evaluation of $f(\cdot)$ for each n-tuple.

A similar problem arises in computational probability when we wish to compute the probability of an event A using the 'law of total probability',

$$\Pr(A) = \sum_k \Pr(A|B_k)\Pr(B_k),$$

where $\{B_k\}_1^n$ is a partition of some space. Here we must generate each B_k of the partition. This method of calculation is called *conditioning*.

In most combinatorial problems, in addition to generating the objects, we must also evaluate some function of the objects, e.g., $f(\cdot)$ and $\Pr(A|B_k)$ above. These evaluations can be expensive and great care must be taken to eliminate unnecessary calculations. It often happens that successive evaluations can be performed rapidly if successive objects are 'close' to each other. If this is the case then a new evaluation is performed by updating a previous (usually the most recent) evaluation. In general updating is much less expensive than complete evaluation. It is for this reason that algorithms which generate objects that are successively close to each other are important. Such algorithms are called *Minimum Change* algorithms.

Good discussions of combinatorial algorithms appear in the books by Even [Eve73], Knuth [Knu73], Reingold, *et al.* [RND78], and Wells [Wel71]. The book by Nijenhuis and Wilf [N&W75] contains an excellent stock of FORTRAN programs for combinatorial problems. The purpose of this note is to add to this stock of programs.

2. A Minimum Change Lattice Point Algorithm

We wish to generate all the lattice points (n-tuples) $x = (x_1, x_2, \ldots, x_n)$, where $l_i \le x_i \le u_i$, $i = 1, 2, \ldots, n$, and l_i, x_i, and u_i are integers. There are $\prod_{i=1}^n (u_i - l_i + 1)$ such points. Furthermore we wish to generate them in such a way that two successive points differ by only one component, i.e., if $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ are successive points then, for some i,

$$x_j = y_j, \quad \forall\, j \neq i \quad \text{and} \quad x_j = y_j \pm 1, \quad j = i. \tag{1}$$

In general there are many sequences whose successive points differ by only one component. Figure 1 shows one minimum-change sequence with $u_1 = 1,\ l_1 = 4,\ u_2 = 1,\ l_2 = 3$ and $u_3 = 1,\ l_3 = 3$

The algorithm we now describe generates points in the same order as shown in Figure (1), which is analogous to binary reflected Gray codes. (The algorithm can be used without modification to generate these Gray codes but is not as efficient as algorithms specially designed for these codes. (See [RND77])).

The generation of the sequence of n-tuples is defined recursively as follows, where $+S(n)$ is the required sequence of n-tuples.

Let

$$+S(1) = \begin{bmatrix} l_1 \\ l_1+1 \\ \vdots \\ u_1-1 \\ u_1 \end{bmatrix} \quad \text{and} \quad -S(1) = \begin{bmatrix} u_1 \\ u_1-1 \\ \vdots \\ l_1+1 \\ l_1 \end{bmatrix} \text{ i.e., the reverse of } +S(1).$$

$$+S(2) = \begin{bmatrix} +S(1), l_2 \\ -S(1), l_2+1 \\ \vdots \\ (-1)^k S(1), u_2 \end{bmatrix}, \quad \text{where} \quad k = u_2 - l_2.$$

Then in general we have

$$+S(n) = \begin{bmatrix} +S(n-1), l_n \\ -S(n-1), l_n+1 \\ \vdots \\ (-1)^k S(n-1), u_n \end{bmatrix}, \quad \text{where} \quad k = u_n - l_n. \tag{2}$$

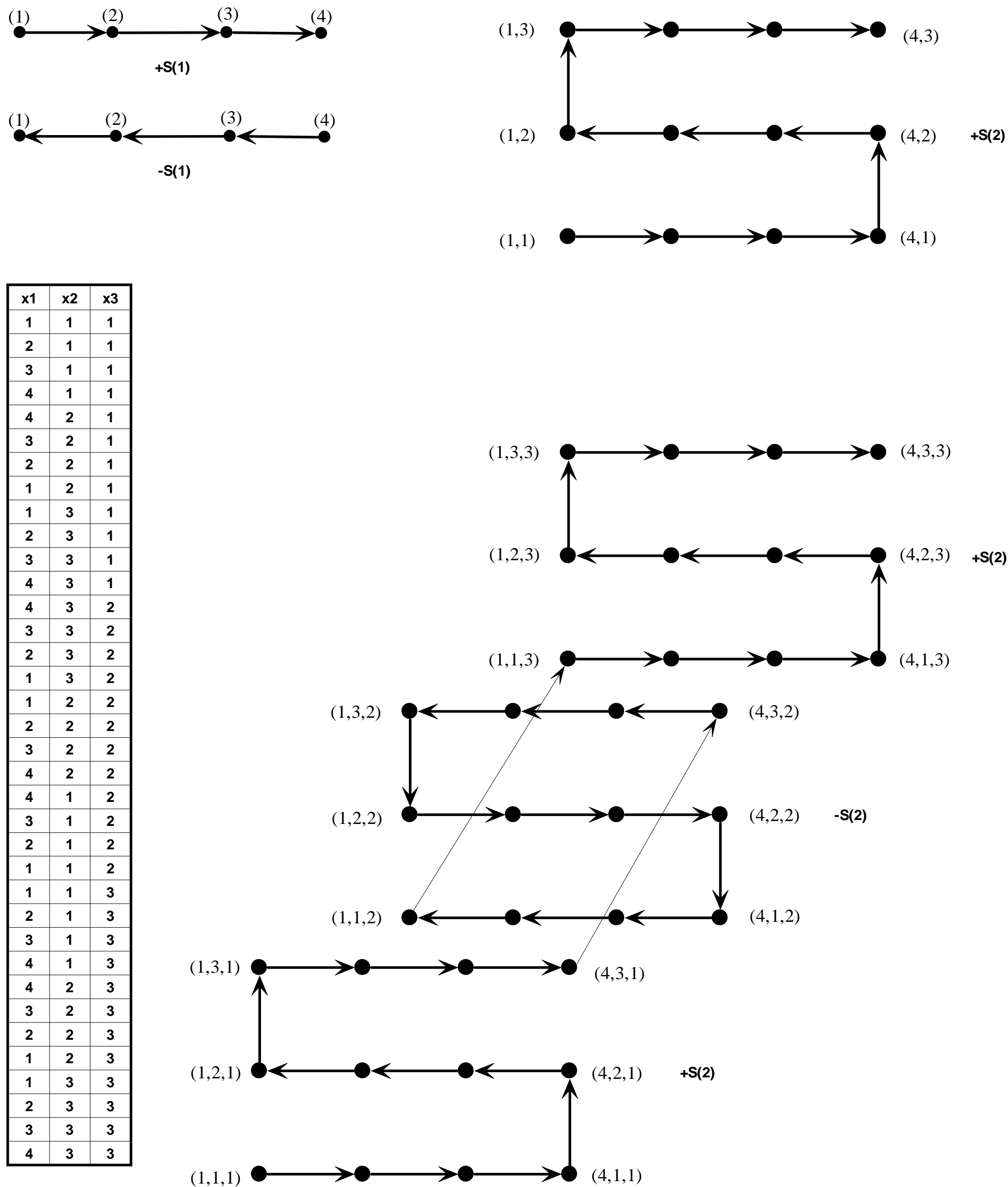
Using the example above, we get

$$+S(1) = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad \text{and} \quad -S(1) = \begin{bmatrix} 4 \\ 3 \\ 2 \\ 1 \end{bmatrix}$$

$$+S(2) = \begin{bmatrix} +S(1),1 \\ -S(1),2 \\ +S(1),3 \end{bmatrix} = \begin{bmatrix} 1,1 \\ 2,1 \\ 3,1 \\ 4,1 \\ 4,2 \\ 3,2 \\ 2,2 \\ 1,2 \\ 1,3 \\ 2,3 \\ 3,3 \\ 4,3 \end{bmatrix}, \quad -S(2) = \begin{bmatrix} 4,3 \\ 3,3 \\ 2,3 \\ 1,3 \\ 1,2 \\ 2,2 \\ 3,2 \\ 4,2 \\ 4,1 \\ 3,1 \\ 2,1 \\ 1,1 \end{bmatrix}, \quad +S(3) = \begin{bmatrix} +S(2),1 \\ -S(2),2 \\ +S(2),3 \end{bmatrix} = \begin{bmatrix} 1,1,1 \\ 2,1,1 \\ 3,1,1 \\ 4,1,1 \\ 4,2,1 \\ \vdots \\ 1,2,3 \\ 1,3,3 \\ 2,3,3 \\ 3,3,3 \\ 4,3,3 \end{bmatrix}$$

Figure 1 shows how this sequence is built from walks in 1-,2-, and 3-dimensional lattices.

© Derek O'Connor, June 25, 2006



x1	x2	x3
1	1	1
2	1	1
3	1	1
4	1	1
4	2	1
3	2	1
2	2	1
1	2	1
1	3	1
2	3	1
3	3	1
4	3	1
4	3	2
3	3	2
2	3	2
1	3	2
1	2	2
2	2	2
3	2	2
4	2	2
4	1	2
3	1	2
2	1	2
1	1	2
1	1	3
2	1	3
3	1	3
4	1	3
4	2	3
3	2	3
2	2	3
1	2	3
1	3	3
2	3	3
3	3	3
4	3	3

Figure 1 : Minimum-Change Walk on a 3-D Lattice

Thus the algorithm generates a sequence of n-tuples from a sequence of $(n-1)$-tuples, starting with a sequence of single numbers. The FORTRAN subroutine below is a non-recursive version of this algorithm.

The flow chart for subroutine is shown in Figure 2. The three conditional statements are labelled C_1, C_2, C_3. The two assignment blocks are labelled B_1 and B_2. The labels L_1, L_2, L_3 denote the number of times the corresponding loops are traversed.

© Derek O'Connor, June 25, 2006

```
  Minimum  Change Lattice Point Algorithm
---------------------------------------------------------------------
              SUBROUTINE Latgen ( L, U, N )
              INTEGER L(1),x(1),U(1),sign(1)
    C         This subroutine generates a sequence of n-tuples
    C         (x(1),x(2),...,x(n)) with l(i)≤x(i)≤u(i)
    C         and successive n-tuples differ by one component.
    C         Initialization
               DO 10 i = 1, n
                  x(i) = L(i)
                  sign(i) = +1
            10 CONTINUE
    C          start of algorithm
               GO TO 40
B1          20 sign(i) = -sign(i)
               i = i + 1
C1             IF(i .eq.  n+1) RETURN
C2          30 IF(x(i) .eq.  L(i) .and.  sign(i) .eq.  -1) GO TO 20
C3             IF(x(i) .eq.  U(i) .and.  sign(i) .eq.  +1) GO TO 20
               x(i) = x(i)+sign(i)
B2          40 CALL Process (x)
               i = 1
               GO TO 30
               END
```

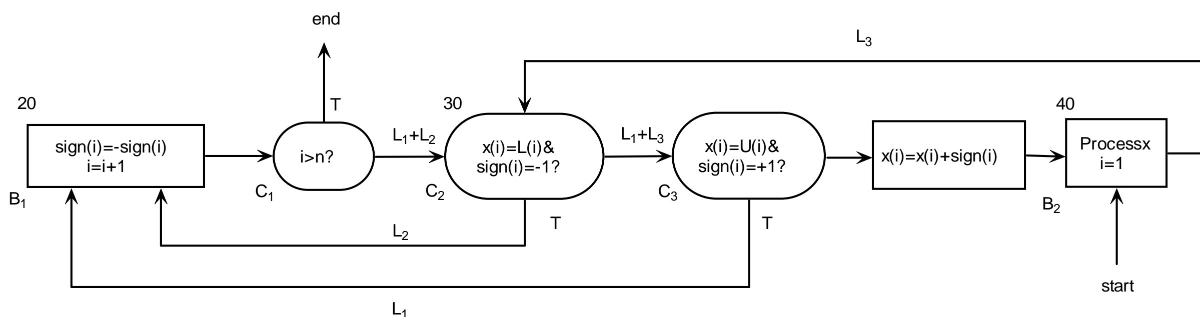


Figure 2 : Flow diagram for `SUBROUTINE Latgen`

3. Analysis of the Algorithm

The algorithm generates all the lattice points in the region $l_i \le x_i \le u_i, \quad i = 1,2,\ldots,n$. Without loss of generality we assume that each $l_i = 1$ and so the number of n-tuples generated is $N = u_1 u_2 \cdots u_n$. The time complexity $T(N)$ of the algorithm must be an increasing function of N and we show below that this function is $O(N)$. A complete analysis requires that we count the number of times each step of the algorithm is executed. Applying Kirchoff's Node Law to the flow chart in Figure 2 we obtain the following statement execution counts:

Table 1: Statement Execution Counts

Statement	B_1	C_1	C_2	C_3	B_2
No. times executed	L_1+L_2	L_1+L_2	$L_1+L_2+L_3$	L_1+L_3	L_3

Assuming the algorithm is correct, then we have $L_3 = N$, the number of sequences generated. We determine the values of L_1 and L_2 by using a tree to represent the execution of the algorithm. Let the root of the tree be at level $n+1$ and let the nodes at level i represent the values taken on by x_i during the execution of the algorithm. Thus for x_1, x_2, and x_3 with $l_1 = l_2 = l_3 = 1$ and $u_1 = 3$, $u_2 = 2$, $u_3 = 4$ we get the tree in Figure 3.

The algorithm starts at the leftmost leaf node with $x_1 = x_2 = x_3 = +1$. It then moves to the right (executing L_3) until C_2 is true. It moves up one level (executing L_2) and then moves to the right (executing L_3) until C_1 is true. It moves up one level (executing L_1) but then must move up to the next level (executing L_2) because C_2 is true.

Thus we see that each move to the right represents an execution of L_3 and each move up represents an execution of either L_1 or L_2. The analysis of the algorithm is now reduced to counting the internal and leaf nodes of the execution tree, because each leaf node corresponds to an execution of L_3 and each internal node corresponds to an execution of L_1 or L_2. The number of leaf nodes is

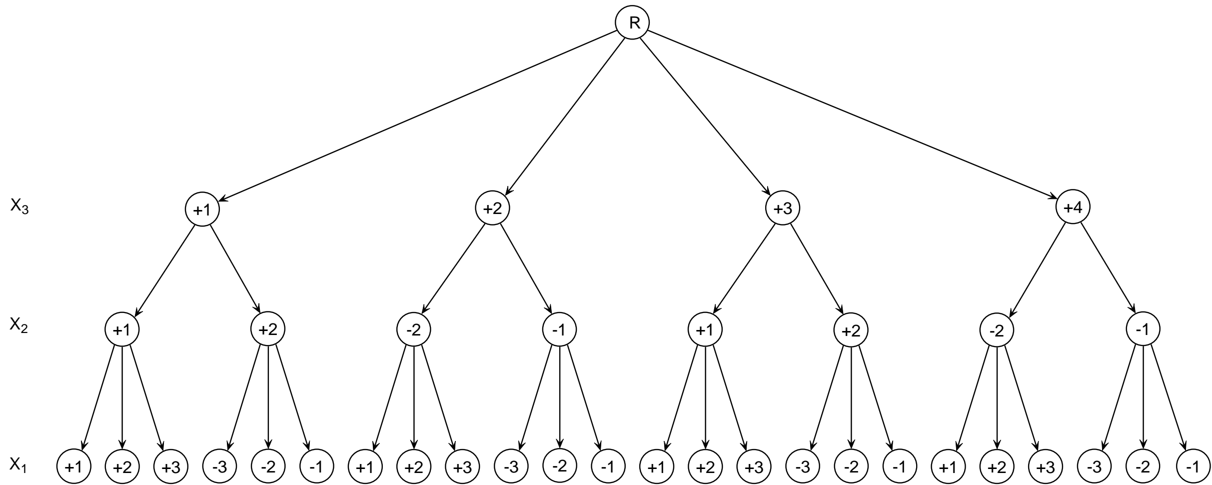


Figure 3 : Execution Tree for subroutine Latgen

$$N = u_1 u_2 \cdots u_n = \prod_{i=1}^{n} u_i \tag{3}$$

The number of internal nodes is

$$M = \sum_{i=2}^{n} M_i = \sum_{i=2}^{n} \prod_{k=i}^{n} u_k, \tag{4}$$

where M_i is the number of nodes at level i.

We have $L_1 + L_2 = M$ and $L_3 = N$. Again, using the execution tree we can see that

$$L_1 = \sum_{i=2}^{n} \lfloor M_i/2 \rfloor, \text{ and } L_2 = \sum_{i=2}^{n} \lceil M_i/2 \rceil.$$

$L_1 = L_2$ if each M_i is even. Each M_i is even if u_n is even. Hence $L_1 = L_2$ if u_n is even and in general $|L2 - L1| \leq n$, with equality holding if $u_2, \ldots, u_n$ are odd. If, for simplicity, we assume $L_1 = L_2 = M/2$, then we get the following operations count, using Table1 and equations (3) and (4)

$2.5M + 2N$	comparisons,
$2M + 2N$	assignments,
$M + N$	additions,
$5.5M + 5N$	total operations.

Assuming each operation requires 1 unit of time, and using (3) and (4) we get

$$\text{Total Time} \quad T(N) = 5\prod_{i=1}^{n} u_i + 5.5\sum_{i=2}^{n}\prod_{k=i}^{n} u_k. \tag{5}$$

Proposition: *$T(N)$ is bounded by a linear function of N.*

Proof: Re-arranging (5) we get

$$\begin{aligned}
T(N) &= 5\prod_{i=1}^{n} u_i [1 + 1.1\left(\frac{1}{u_1} + \frac{1}{u_1 u_2} + \cdots + \frac{1}{u_1 \cdots u_{n-1}}\right)] && (6)\\
&\leq 5\prod_{i=1}^{n} u_i [1 + 1.1\sum_{i=1}^{n-1}\frac{1}{2^i}], \text{ for all } u_i \geq 2 \\
&= 5\prod_{i=1}^{n} u_i [-0.1 + 1.1\sum_{i=0}^{n-1}\frac{1}{2^i}] \\
&= 5\prod_{i=1}^{n} u_i [-0.1 + 2.2\left(1 - \frac{1}{2^n}\right)] \\
&< 5\prod_{i=1}^{n} 2.1 u_i \\
&= 10.5\prod_{i=1}^{n} u_i = 10.5N && (7)
\end{aligned}$$

Hence $T(N) < 10.5N$, where N is the number of n-tuples generated. □

4. Speeding Up the Algorithm

The expression (6) for $T(N)$ contains the term $1/u_1 + 1/u_1 u_2 + \cdots + 1/u_1 u_2 \cdots u_{n-1}$. The value of this term is minimized if the us are permuted so that $u_1 > u_2 > \cdots > u_n$. For example, with $u_1 = 2,\ u_2 = 3,\ u_3 = 4,\ u_4 = 5,\ u_5 = 6$ we get $N = 720$ and $M = 516$ which gives a total of 6438 operations. With $u_1 = 6,\ u_2 = 5,\ u_3 = 4,\ u_4 = 3,\ u_5 = 2$ we get $N = 720$ and $M = 152$ which gives a total of 4436 operations, a savings of 30%.

© Derek O'Connor, June 25, 2006

Table 2: Tests of Latgen Algorithm

Test	u_1	u_2	u_3	u_4	u_5	u_6	u_7	u_8	u_9	u_{10}	L_1	L_2	L_3	$T(N)/N$
1	2	3	4	5	6						256	258	720	9.0
2	6	5	4	3	2						76	76	720	6.2
3	7	5	4	3	3						113	113	1,260	6.0
4	2	2	3	3	4	4	5	5	6	6	225,561	225,561	518,400	9.9
5	6	6	5	5	4	4	3	3	2	2	52,227	52,227	518,400	6.0
6	11	10	9	8	7	6	5	4	3	2	2,018,956	2,018,956	39,916,800	5.6
7	2	2	2	2	2	2	2	2	2	2	524,287	524,287	1,048,576	10.5

Sorting the us in descending order has the effect of minimizing the number of internal nodes in the execution tree, for a given number of leaf nodes. This also explains the bound in (6) : a complete binary tree has the maximum number of internal nodes for a given number of leaf nodes which gives $T(N) = 10.5N$.

5. Testing the Algorithm

The program given in Section 2 was metered and tested to verify the analysis of Section 3. Seven tests were run with counters in the three loops L_1, L_2, and L_3. The ratio $T(N)/N$ was calculated to show how the slope of $T(N)$ varies with the input data. It can be seen that $T(N)$ is bounded by $10.5N$ and that this bound is achieved in Test 7, the Binary Tree case.

Tests 4 and 5 clearly show the savings (40%) that are achieved by re-ordering the us in descending order.

6. References

[Wel71] Wells, M.B, *Elements of Combinatorial Computing*, Pergamon Press, (1971).

[RND77] Reingold, E.M., Nievergelt, J., and Deo, N., *Combinatorial Algorithms : Theory and Practice*, Prentice-Hall, (1977).

[Eve73] Even, S., *Algorithmic Combinatorics*, Macmillan, (1973).

[Knu73] Knuth, D., *The Art of Computer Programming, Vol.3*, Addison-Wesley, (1973).

[N&W] Nijenhuis, A., and Wilf, H.S., *Combinatorial Algorithms*, Academic Press, (1975).

© Derek O'Connor, June 25, 2006