Journal of Statistical Physics, Vol. 58, Nos. 3/4, 1990

The Impact of Large-Scale Computing on Lattice Statistics

J. L. Martin[1]

Received July 14, 1989

The use of computers in theoretical physics has grown dramatically over the years; this is as true in lattice statistical mechanics as anywhere. This paper is concerned with one such aspect with which the name of C. Domb has been closely associated: the enumeration of embeddings of connected structures in an unlimited crystal lattice. An informal account is given of a recent computer project originating in the combinatorial "shadow" method developed by M. F. Sykes: the determination of the numbers and the properties of *cluster* embeddings in crystal lattices. Sykes' approach has opened a way to information which was earlier considered to be forever beyond reach. The principles are given and the algorithms sketched; the detailed FORTRAN programming is not given. The methods used have had to be specially developed, but some have a wider application for computer algebra when the computational task is massive. Provided the computer is large enough and fast enough, impressive results may be obtained in return for a reasonable effort. In practice, this implies that the computer may have to be one of the largest and fastest, or else that it is dedicated to the task.

KEY WORDS: Lattice statistics; cluster enumeration; combinatorics.

1. GRAPHICAL EXPANSIONS

Power series expansions—and indeed expansions in more general functions —appear frequently in physics. Most often they arise as the result of a perturbation procedure: an entity of physical interest is expanded in powers of a parameter which is usually taken to be small; a relatively short series may be all that is needed. Sometimes, however, a much more ambitious goal is set, a goal with which the name of Prof. C. Domb (see, e.g., ref. 1) has long been associated: A power series is used to estimate the location and nature of a *singularity* of a function, such as a critical point singularity of a ther-

[1] Wheatstone Physics Laboratory, King's College, Strand, London WC2R 2LS.

modynamic function; for this purpose it is essential that *as many terms as possible* of the series should be available. This paper is concerned with the development of lengthy series with critical point applications in mind.

In some cases the form of the successive terms in a perturbation series becomes unmanageably complicated when written out in conventional algebraic notation. This is particularly true for the example of quantum electrodynamics, and Richard Feynman invented the shorthand of the *graphical expansion* in which the contribution at each order is represented as the sum of a collection of *diagrams* or *graphs*; each graph is to be interpreted as a multiple integral according to well-defined rules. Mayer's cluster integrals of a decade earlier may be given a similar representation. Since that time it has become clear that graphical expansions are of value for mastering the complexity that may arise in other contexts.

Such a context is *lattice statistics*: the statistical mechanics of a set of systems located at the sites of one of the usual (infinite) crystal lattices, with more or less local mutual interactions. Here Feynman "integrals" are *sums*, as a consequence of the discreteness of the lattice structure; moreover, if the range of the interaction between sites is strictly finite, the sums are also strictly finite and the evaluation of each term of a perturbation expansion becomes an *enumeration problem*. In most cases, the enumeration problem reduces to a sequence of enumerations of *the possible embeddings of topological structures of specified construction* in the crystal lattice of interest. We shall call this a *counting problem*.

My own deep involvement with counting problems dates from the early 1960s and arose from chance encounters with Prof. M. E. Fisher and with Prof. C. Domb. At the time I was engaged in work of a quite different nature, but I was intrigued by their statement to the effect that the essentially combinatorial methods which were needed to make sensible progress in any realistic counting problem were far too subtle to be enshrined in a computer program. Of course, those were the days when computers were generally regarded as number machines and not as the superbly versatile devices of today. It took over a week, without benefit of high-level language or operating system, to develop an elementary program to count self-avoiding walks(2),[2]; nowadays it can be done in 15 min. The computer—the National Physical Laboratory ACE—had roughly 800 words of "fast" memory (the latency time was about $\frac{1}{2}$ msec). How far we have come in such short time!

The project really began to get under way with the arrival of the English Electric KDF9 computer, with the capacity and speed of a small modern micro. It surprises me in retrospect how much was achieved with

[2] See ref. 3 for another example of very early computer enumeration.

such meager resources. However, the theme of this paper is modern: an account will be given of a typical large-scale computer onslaught on a counting problem of massive proportions which has required the use of one of the fastest and largest computers currently available, and the aim will be to show how impressive the results can be, when the approach is right and the tools are available.

2. CLUSTER-COUNTING AND THE SYKES METHOD

There are many types of counting problem. One which is of great interest for many applications in statistical physics is the enumeration of clusters. A *connected cluster* is defined—for our purposes—as a set of sites of our chosen lattice which, when taken with all lattice bonds connecting pairs of these sites, yields a topologically connected structure. Here we are concerned with the design of effective computer algorithms for the enumeration of connected n-site clusters; as a particular instance, it illustrates well the multifaceted approach which is needed for a challenge of this kind.

It is indeed a challenge. The total number of n-site clusters on a lattice of coordination μ may be expected to grow roughly as μ^n as n increases (the worst case of usual interest is the fcc lattice, for which $\mu = 12$). Any computer program which can make much impression in the presence of exponential growth will need to be efficient, and will be expected to use techniques for fast recovery of information.

Experience has shown that the best approach—when available—is one which supplements brute-force enumeration with algebraic procedures based on combinatorial theorems, usually expressed in terms of *generating functions*. Theorems are hard to come by in general; however, Sykes(4) has developed a combinatorial approach which replaces an impossibly large enumeration by a very large (but possible) enumeration followed by a very large (but possible) algebraic manipulation. In this way, the size of cluster which can be enumerated is substantially increased. The growth of the counting problem is still exponential—though with a smaller ratio—and the presence of a natural cutoff therefore persists. The algebraic manipulation grows relatively slowly at first, but the ultimate growth is *faster* than any exponential and there is a hard upper limit to what can be achieved.

The Sykes method applies to bipartite lattices, and is the subject of the rest of this paper.

The sites of a *bipartite lattice* fall into two disjoint exhaustive sets **X** and **Y** such that the two sites at the ends of any bond of the lattice belong one to each of the sets. (The plane square lattice with nearest neighbor bonds is bipartite: think of the black and white squares of a chessboard. The plane triangular lattice is not bipartite.)

Strictly the Sykes method applies to any bipartite network. However, the most useful outcome occurs when the network is symmetric against interchange of the sets **X** and **Y**. The "usual" even regular crystal lattices have this property. The lattice is taken to be of indefinite extent in all directions; thus, all cluster counts are to be taken as counts per site, or perhaps counts per **X**-site, as convenient.

Define the *enumerator* F as the generating function

$$F(x, y) = \sum_m \sum_n x^m y^n c_{mn} \tag{1}$$

in which c_{mn} is the count of $(m+n)$-clusters involving exactly m (n) sites from the set **X** (**Y**). Each such cluster C may be regarded as the union of two partial clusters C_X and C_Y, of m and n sites, respectively. The summations in F are partially broken down with regard to all the possibilities for C_X,

$$F(x, y) = \sum_{C_X} x^{m(C_X)} E(y; C_X) \tag{2}$$

in which $m(C_X)$ is the number of sites in C_X, the summation is over all C_X which can give rise to a connected cluster C (the *relevant* C_X), and

$$E(y; C_X) = \sum_n y^n c_n(C_X) \tag{3}$$

Here $c_n(C_X)$ is the total number of ways of adding n Y-sites to the partial cluster C_X to yield a *connected* cluster C.

Sykes' achievement has been to obtain an *algebraic* route to the enumerator E, given only the cluster C_X, based on a combinatorial argument reminiscent of "inclusion-exclusion." This suggests a computational procedure in which the partial clusters C_X are listed, the corresponding E for each is evaluated, and the function F is progressively assembled.

Such a procedure would still be infeasible were it not for the fact that in general several (sometimes very many) distinct partial clusters C_X lead to the *same* enumerator E. Detailed knowledge of how the partial cluster falls on the lattice is not at all necessary: what we need is the *Structure* of the partial cluster—always with the initial capital to indicate its special nature.

A typical four-point X-cluster on the square lattice is shown in Fig. 1, along with its Structure; the latter consists of the four X-points (the *Points*) taken with *every* neighboring Y-point (the *Neighbors*), ten in all. (In this case, this same Structure arises from eight distinct X-clusters on the square lattice, so that the condensation is considerable even at this early stage.)

We need to codify Structures in some way. It is useful to use Sykes' picturesque language: the Points of the Structure are to be thought of as casting *shadows* on the Neighbors. A *representative* of a Structure is a list of the ways in which the shadows fall on the Neighbors; for our example, with the Points labeled as shown, the representative is the collection of ten *Shadows*, one for each Neighbor,

$$A \quad A \quad C \quad D \quad D \quad AB \quad AB \quad BC \quad CD \quad BCD$$

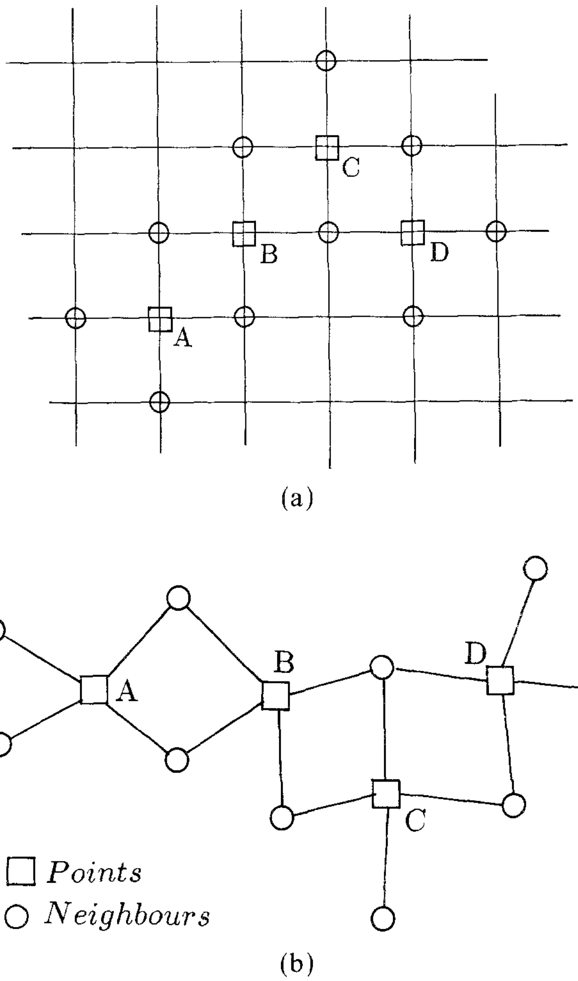

Fig. 1. (a) A typical 4-Point X-cluster on the square lattice, (b) Its Structure. (□) Points, (○) Neighbors.

(See also Fig. 5.) When the coordination of every Point is the same (as is true for the case of an infinite crystal lattice), those Neighbors shadowed by only one Point may be safely omitted from the list: their Shadows can be reconstituted from a knowledge of the others. Thus we are left with an abbreviated representative

$$AB{:}\,2 \quad BC{:}\,1 \quad CD{:}\,1 \quad BCD{:}\,1$$

As they come to be computed, X-clusters are to be classified by Structure, not merely by representative. The labeling used is not of the essence, and relabeling the Points of a Structure may yield a *different* representative for that same Structure; there may therefore be as many $m!$ representatives for a single m-Point Structure. The challenging problem of how to recognize the equivalence or otherwise of two different representatives must at some point be faced.

The expression for F may now be rewritten as a sum over Structures,

$$F(x, y) = \sum_S x^{m(S)} E(y; S)\, N(S) \tag{4}$$

in which $m(S)$ is the number of X-sites in a partial cluster with Structure S, $E(y; S)$ is the enumerator computed for this Structure, and $N(S)$ is the total number of partial clusters C_X with this Structure. So we are led to the computational procedure:

(i) Generate all relevant partial X-clusters of a chosen fixed size m, and "pigeonhole" each according to its Structure; in this way, we obtain the enumeration $N(S)$ for each possible Structure S.

(ii) For each Structure S which has appeared in the course of process (i), evaluate $E(y; S)$.

(iii) The above is to be done for $m = 2$ upward, as far as the constraints of computer time and storage will allow. (The case $m = 1$ is trivial.)

The enumerator F is then trivially assembled [Eq. (4)], to the order permitted by the extent of the computation. When F is known to be symmetric against interchange of x and y (as always in actual applications), more of the series may be filled in by simple transcription. In the symmetric case, if the maximum feasible value of m is M, then full information on connected clusters of size $2M + 1$ can be obtained in this way.

3. COUNTING CLUSTERS ON A LATTICE

General-purpose computer programs for enumerating clusters on networks have been with us for at least 20 years, and the techniques have steadily improved. A fairly formal account—perhaps slightly more involved than it should have been—has been given elsewhere,[5] so we shall be content here to outline the main principles with the help of a simple illustration.

For this purpose let us consider the enumeration of *polyominoes.*[6] By analogy with a domino (which ought I suppose to be diomino in this context), a polyomino is a plane connected arrangement of a finite number of square tiles. (Two tiles are connected if they have one edge in common; general connectivity follows by induction.) An example is shown in Fig. 2. This figure also illustrates a crucial point: the polyomino is embedded in a *half*-plane with a step in the boundary. We take it to be evident that any polyomino in general position may be *uniquely translated* so that one of its tiles will occupy the position marked **1**. The polyominoes will be counted by a process of growth from a single seed-tile; demanding that the seed-tile should be the unique tile at **1**, and no other, cuts the total count by a factor of precisely n for an n-point polyomino. This requirement is easily extended to more general contexts, and can be trivially implemented in a computer program.

The growth process is illustrated in Fig. 3 for the six polyominoes of three tiles. The seed-tile is provided with a *halo* of those locations—marked **2** and **3**—where further tiles may be placed so as to maintain the connectedness. (There are two such locations, not four; remember the step of Fig. 2.) Adopting location **2** for the next tile extends the halo with two further locations, labeled **4** and **5**—in no particular order. Clearly, loca-

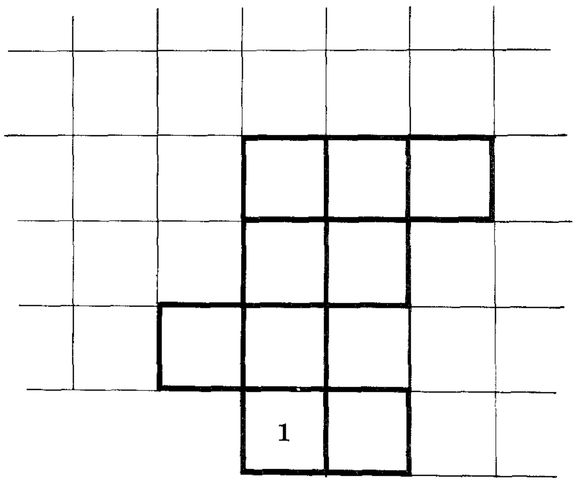

Fig. 2. A correctly positioned embedding of a cluster in the square lattice half-plane.

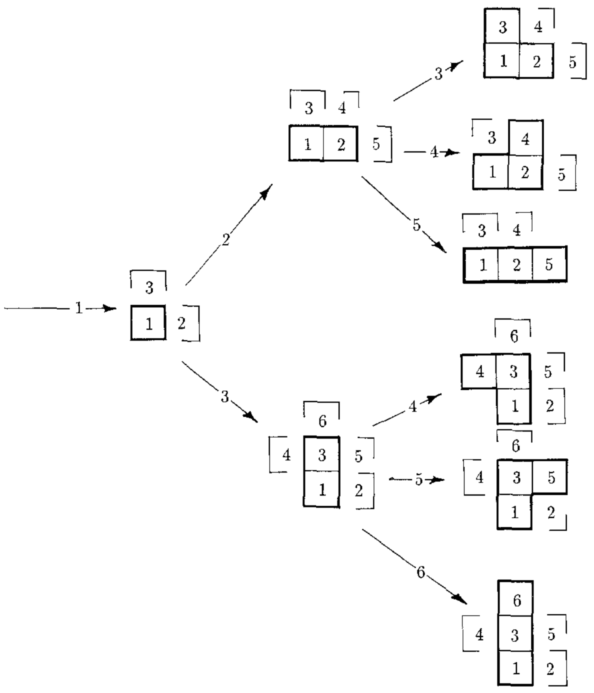

Fig. 3. The polyomino growth-tree.

tions **3**, **4**, and **5** are now available for the third tile; in this way we arrive at the first three 3-polyominoes. (Of course, the halo of the final tile is an irrelevance, and we ignore it.)

Now adopt location **3** for the second tile, extending the halo with three locations, **4**, **5**, and **6**. The halo now comprises four locations, but using location **2** for the third tile will clearly result in a repetition; this location is now prohibited for growth. If the new halo locations are labeled consecutively in the natural way, as illustrated here, the rule is very simple: *the label of each new tile must be greater than the label of its predecessor.* The label sequences for the nine polyominoes in Fig. 3, in the order of appearance, are

1 12 123 124 125 13 134 135 136

and these clearly satisfy the rule.

Generating clusters in this way is very efficient, since a large majority of clusters have much in common with their immediate predecessors, and minor changes are usual in passing from one to the next. The implementation of the process as a fast computer program is straightforward; the algorithm is naturally recursive, however, as may be expected from the family-tree structure of Fig. 3, and does not fit too easily into a naturally nonrecursive language like FORTRAN.

The present application calls for the enumeration of partial clusters on the X-sublattice of an even lattice. The algorithm to be used is identical; the only change is that the lattice vectors must be replaced with the vectors for the *next-neighbor* lattice, obtained as the set of all differences of pairs of vectors of the original lattice. Figure 4 illustrates this for the square lattice.

4. DETERMINING THE STRUCTURE OF AN *X*-CLUSTER

As each X-cluster is revealed by the process of the last section, its Structure—or rather, one of the representatives of its Structure—must be worked out. It is not acceptable to work out the Structure from scratch for every new cluster; this would entail much repetition of work already performed and would consequently be far too inefficient. However, because each cluster is built up point by point, it is possible to assemble the Structure-representative by incremental adjustments as the cluster itself grows.

Each new point of the X-cluster is a new Point of the Structure. Some of its Neighbors must already be part of the Structure; their Shadows need to be adjusted. Some of its Neighbors may be new to the Structure; their Shadows will appear in the *full* representative for the first time, though we shall avoid including them in the abbreviated representative until they are influenced by at least *two* Points.

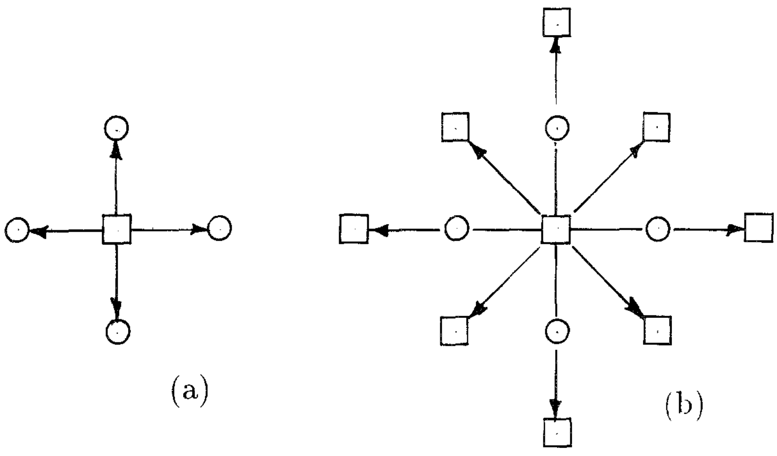

Fig. 4. (a) Square lattice neighbor vectors, (b) The derived next-neighbor vectors.

When an X-cluster reaches full size, it is vitally important that the representative should be very quickly available for archiving. This part of the algorithm is visited once for every cluster generated, and the preparation and archiving processes must be made as efficient as possible. To facilitate speedy preparation, redundant information is retained as the representative is built up. Archiving ("pigeonholing") is discussed in Section 5.

To fix ideas, we shall again refer to the example of Fig. 1. The cluster generator will deliver the Points one by one, let us say in the order *A B C D*. Building up the representative requires that the (relatively few) Neighbors of each new Point be examined, and their new or altered Shadows listed. Additionally—and redundantly—the program maintains a tableau of *how many* Shadows of each kind there are, and a further record is kept of pointers to the tableau-entries which are relevant for inclusion in the abbreviated representative. (Computer programmers will be familiar with the need for such redundancy in applications of this kind.) By reference to Table I and Fig. 5, it should be clear how the redundant information can be efficiently adjusted as a cluster is built up, and how the pointer record yields the desired final result. There is a caution: it is possible for a nonzero entry in the tableau to fall back to zero at a later stage in the generation of the cluster; the corresponding item in the record must then be ignored when the completed representative comes to be archived. Marks—shown by arrows—are kept to indicate how many Neighbors must be erased when clusters come to be dismantled by the cluster generator.

A program constructed as described so far will generate a stream of Structures—more precisely, a stream of abbreviated representatives—one for each X-cluster of interest. In a sense this solves the counting problem. However, the result is utterly unsurveyable if left like this—the sequence of representatives may be so colossal that *immediate* action must be taken to condense it. Typically, a randomly selected representative will occur very many times in the sequence, arising as it may from X-clusters with different shapes and yet with the same Structure. It is necessary to "pigeonhole"

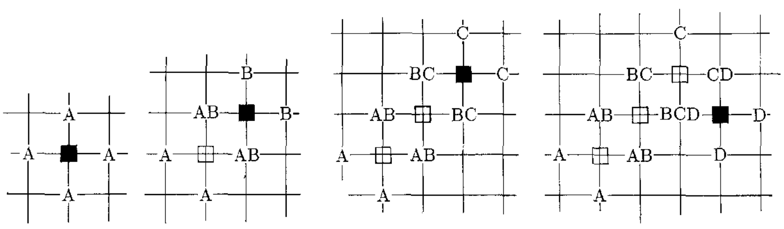

Fig. 5. Adjustments to the Shadows as a typical X-cluster grows.

Table I. Building Up a Structure Representative

After Point *A* at position (0, 0):

Neighbors:	(1, 01)	(0, 1)	(−1, 0)	(0, −1)
Shadows:	*A*	*A*	*A*	*A*

Nonzero tableau: (*A*) = 4
Pointer record: empty

After point *B* (1, 1):

				↓ [mark (2)]		
Neighbors:	(1, 0)	(0, 1)	(−1, 0)	(0, −1)	(2, 1)	(1, 2)
Shadows:	*AB*	*AB*	*A*	*A*	*B*	*B*

Nonzero tableau: (*A*) = 2 (*B*) = 2 (*AB*) = 2
Pointer record: *AB*

After Point *C* (2, 2):

				↓		↓ [mark (3)]		
Neighbors:	(1,0)	(0, 1)	(−1, 0)	(0, −1)	(2, 1)	(1, 2)	(3, 2)	(2, 3)
Shadows:	*AB*	*AB*	*A*	*A*	*BC*	*BC*	*C*	*C*

Nonzero tableau: (*A*) = 2 (*C*) = 2 (*AB*) = 2 (*BC*) = 2
Pointer record: *AB BC*

After Point *D* (3, 1):

				↓		↓		↓ [mark (4)]		
Neighbors:	(1,0)	(0, 1)	(−1, 0)	(0, −1)	(2, 1)	(1, 2)	(3, 2)	(2, 3)	(3, 0)	(4, 1)
Shadows:	*AB*	*AB*	*A*	*A*	*BC*	*BCD*	*C*	*CD*	*D*	*D*

Nonzero tableau: (*A*) = 2 (*C*) = 1 (*D*) = 2 (*AB*) = 2 (*BC*) = 1 (*CD*) = 1 (*BCD*) = 1
Pointer record: *AB BC BCD CD*

each new representative on arrival along with its identical predecessors; a representative without an identical predecessor will call for the creation of a further pigeonhole to accommodate it. When the process is complete, we shall be left with a set of pigeonholes, each labeled with a representative and containing the *count* of the appearances of that representative in the sequence.

Even this is not enough. Two representatives which differ on account only of a relabeling of the Points of the *X*-cluster belong to the same Structure; on this account, some Structures may possess thousands of distinct representatives, and ambitious counting calls for a further condensation (*canonization*: see Section 7). Canonization entails a permutation process of some kind, and needs careful design. In any case, it is postponed as far as feasible: the sequence is pigeonholed according to representative in the first instance, and is condensed by canonization at the end, or whenever a shortage of storage space makes this necessary.

5. THE PIGEONHOLE ALGORITHM

Cluster-counting entails the computer generation, without omission or repetition, of the very large number of clusters of a specified set; frequently the clusters are to be *subclassified* according to some feature or other: in the current application the relevant feature is the Structure.

Later we shall need to perform combinatoric feats by large-scale algebraic manipulation. When two algebraic expressions are multiplied together, the result in the first instance is a possibly very large number of terms (the pairwise products) which will need to be *reduced* by summing those terms which are identical apart from coefficient. This "collecting of terms" is a process which is essentially identical to the subclassification of the last paragraph. It is the bane of algebra, whether manual or automated.

It is perhaps true to say that none of this matters in everyday circumstances. However, in the current application we expect to have to deal with the generation of vast numbers of clusters, each one of which needs to assigned unerringly and quickly to the correct Structure, followed by the generation of vast numbers of terms as the combinatorial theorems are applied. In such circumstances, it is essential to make the process of subclassification (of either kind) as efficient as ingenuity will allow.

First, an informal example to illustrate the rules. Imagine that some algebraic process or other delivers a stream of terms to be summed, such as

$$2x,\ -3y, 4xy, 5x, 3y, 0z,\ldots$$

The partial sums are clearly

$$0;\ \ 2x;\ \ 2x-3y;\ \ 2x-3y+4xy;\ \ 7x-3y+4xy;\ \ 7x+4xy;\ \ 7x+4xy;\ldots$$

It is useful to recast this in a more general-purpose notation:

	initially:	$[\ \ ]$
(1)	adding $\{x:2\}$ gives	$[\{x:2\}]$
(2)	adding $\{y:-3\}$ gives	$[\{x:2\}\{y:-3\}]$
(3)	adding $\{xy:4\}$ gives	$[\{x:2\}\{y:-3\}\{xy:4\}]$
(4)	adding $\{x:5\}$ gives	$[\{x:7\}\{y:-3\}\{xy:4\}]$
(5)	adding $\{y:3\}$ gives	$[\{x:7\}\{xy:4\}]$
(6)	adding $\{z:0\}$ has no effect	
(7)	$\cdots$	

In general, a *shelf* $[\cdots]$ is a set of *pigeonholes* {name : content}, where the *name* may be thought of as a label attached to the pigeonhole and *content* is what is currently to be found inside.

A shelf is built up from a stream of *terms* {name : content}. (The notations for *terms* and for *pigeonhole* are similar, without ambiguity). The rules for *shelving* each incoming term are straightforward:

(i) If the name of the new term does not match the name of any pigeonhole, then create a new pigeonhole to accommodate the term (steps 1, 2, 3 of the informal example).

(ii) If the name of the term matches the name of a pigeonhole, then augment the content of that pigeonhole by the content of the term (step 4).

(iia) If additionally the augmented content is now empty, then abolish the pigeonhole (step 5).

(iii) Terms with empty content are to be discarded (step 6).

The informal example shows how pigeonhole is applied to algebra: the *name* is the form of the monomial, and the *content* is the coefficient; the *shelf* contains the expression as a whole. In the case of cluster-counting, each new cluster will possess a Structure, and will generate a new term {Structure: 1}, in which the *name* is the Structure, and the *content* is 1 (to count one new cluster, naturally). The shelf ultimately contains the detailed enumeration of the clusters classified by Structure. [Of course, in this instance rules (iia) and (iii) are not called upon.]

The triviality of the idea conceals the appalling inefficiencies which may arise if the mass of information is badly handled. The nub of the difficulty is this: the typical collection of names which may arise—in either type of application—is expected to be very irregular, and in general there is no straightforward natural way of assigning the pigeonholes to the elements of an array, unless the array is to be so sparsely used that even the largest computers are far too small to hold it. Other styles of storage must therefore be used. Hash-addressing (see, e.g., ref. 7) recommends itself as a programming technique, and indeed it plays a crucial part in the algorithm. However, with a virtual-memory computer, or in circumstances where the information is so massive that disc storage has to be used, hash-addressing can take one only so far, and supplementary procedures must be invoked.

6. THE PROGRESS OF A TERM THROUGH THE SHELVING ALGORITHM

Pigeonholing has at least two applications: counting Structures and collecting terms in an algebraic expression. Both are relevant to the work of this paper. A potentially troublesome algorithm ought never to be programmed more than once, and the routines described below have been designed as a library package so as to be available in wider contexts, such as general-purpose algebra.

The shelving of a term of the sequence proceeds in two phases, as follows.

Phase One is designed to be as fast as possible. A region of memory is set aside as a temporary *receptacle* for incoming terms. Well-understood standard hash-address techniques are used: the name of the term is "pseudorandomized" to yield a pointer into the receptacle, and the following procedure is carried out:

```
[*]   if the pointer locates the correct pigeonhole
        then conflate the new term into this pigeonhole and quit
      if the pointer locates no pigeonhole
        then set up a fresh pigeonhole for the new term and quit
      otherwise advance the pointer by one place and go back to [*]
```

This procedure determines whether the name of an incoming term is new to the receptacle or not. If new, then a new pigeonhole is formed; otherwise the term is absorbed into the already existing pigeonhole of the same name. Clearly such a process deals adequately with rules (i) and (ii); application of the other rules is deferred. With care, the method may be made very fast.

Phase One would be enough to cope with manipulations of moderate extent. However, it may be necessary to restrict the size of the receptacle for one reason or another. The physical memory size of the computer may be too small to accept all the information, which will need to spill out to the disc or tape, or the page-faulting of a virtual-memory computer may be destroy the advantages of speed if the receptacle is made too large. In addition, whatever the size of the receptacle, it may happen that "conflicts," when the pointer fails to locate the correct pigeonhole quickly enough, may become unacceptably frequent. For one reason or another, therefore, it may be necessary to *decant* the contents of the receptacle, and to start again.

The act of decanting introduces the information to Phase Two. The pigeonholes in the receptacle are removed and presented one by one as terms in their own right to a sorting algorithm (the sort is carried out on pigeonhole-name); any empty pigeonholes are filtered out at this stage. (It is here that canonization of the representatives is performed; at this late stage it needs to be done far less frequently, but it cannot be postponed further.) Terms with the same name become immediate neighbors as a result of the sort, and the routine is extended to conflate such neighbors into a single term—and also, if the resulting content is empty, to discard it altogether. The sole reason for the ordering is to make this conflation possible; it would be difficult to find a more efficient method.

When all items have been decanted, Phase One is resumed with an empty receptacle and the cycle repeated until the sequence of terms comes to an end. The shelf is then available as an ordered list.

The sorting method is an "$n \log n$" sort–merge algorithm in an ingenious variant due to A. L. J. Wells (unpublished). It is in no way inferior in efficiency to the more familiar algorithms, but it has the extremely useful feature of being *incremental*: most of the sorting is carried out as the items arrive and the resulting conflations relieve what might otherwise be acute pressure on computer memory. (The embodiment as a computer program incorporates a "garbage collector" to organize the storage released by conflation. Additionally, much use is made of pointers for the sake of time efficiency. This is not the place to do more than merely mention these matters.)

7. CANONIZATION

When X-clusters are being enumerated, the end product is to be a shelf each of whose pigeonholes yields one of the coefficients $N(S)$ in Eq. (4); indeed, $N(S)$ will be represented by the pigeonhole $\{\rho(S) : N\}$ in the style of Section 5, in which $\rho(S)$ is the canonical representative (now to be defined) of the Structure S. The entire sum of Eq. (4) will then be covered by the shelf as a whole.

The representative of a Structure is a collection of Shadows, p in all, say. Its formulation relies on some labeling or other of the m Points of the Structure; this labeling is not of the essence, and two representatives which differ only in that the labelings which produce them are different must be treated as identical. This leads to a problem of identification which may become acute: the total number of representatives associated with one Structure may be as great as $m!\,p!$. (The first factor is associated with the labeling, the second with the order in which the Shadows are listed. It is the first factor which causes the real problems.)

The natural way to meet this challenge is to designate just one of the many representatives of a Structure S as the *canonical representative* $\rho(S)$ and to provide an algorithm for transforming any representative into its canonical counterpart. A thoroughly impracticable approach would be to form lists of the representatives, with the canonical version at the head of each. A better approach is to *imagine* such lists, each arranged in order according to some prearranged rule, and to apply the relevant permutations to the incoming representative, noting any version which is—

according to the rule—"earlier" than any previous one; in this way, the unique representative at the head of the imagined list is straightforwardly identified.

This works well up to a point. However, the aim is to design routines which will work for Structures with up to 12 Points or thereabouts; since the number of Point-labelings is then 12! =479001600, any way to reduce the labor must be welcomed. Sometimes the Structure itself will provide an answer; in the example of Fig. 1, list the numbers of Neighbors of each Point with different "intensity" of Shadow:

Shadow intensity	1	2	3	4 ...
Point A	2	2		
B		3	1	
C	1	2	1	
D	2	1	1	

Clearly, the immediate environments of the Points are all different in this case: so map the environments on to the integers in some way, place the integers in order of magnitude, and thus establish a canonical labeling of the Points. The canonical representative is the representative which results from the canonical labeling.

At the other extreme, there are Structures whose every Point has the same immediate environment: certain kinds of closed loop, for example. The above process will then do nothing toward canonization, and generating all permutations of the labels may be prohibitively expensive. It may be worth giving these Structures some alternative treatment. Loops, for example, are easily recognized. Other "difficult" Structures occur rarely, and it may be simplest to raise a flag when they appear, and to accept their uncanonized representative as adequate.

Most Structures lie between the extremes: the Points are classified into sets according to environment, and labeling the Points in order of environment yields a partial canonization only. However, all that is now required is to apply only those permutations which do not disturb the already established sequence of environments, and this task is usually orders of magnitude faster than the full permutation.

A transparent method of working through all $n!\cdot$permutations of n objects goes as follows. Imagine that the objects are located in boxes 1, 2, 3,.... The symbol (12) means "exchange the objects currently in boxes 1 and 2"; similarly for the others. The starting configuration is Permutation 0.

The rule to obtain Permutation s from Permutation $s-1$ is:

if s is odd, perform (12);
if s is even, but not divisible by 6, then (23);
if s is divisible by 6 but not by 24, then (12) (34);
if s is divisible by 24 but not by 120, then (23) (45);
if s is divisible by 120 but not by 720, then (12) (34) (56);...

The general form should be clear. This rule takes us up to and including Permutation $(n!-1)$ without omissions or repetitions. (Note that all the exchanges are made between *adjacent* boxes. It is simple to arrange the internal computer representation so that the objects which we desire to permute are adjacent binary digits in the components of the representative of a Structure. If the above permutation rules are used, then the process will require the exchange of adjacent digits, and it transpires that the passage from one permutation to the next can be achieved by a single FORTRAN statement.)

Moreover, suppose that examining the environments of the Points of some given Structure has shown, for example, that we need to permute the contents of locations 123 and of 45 and of 67 separately, in all combinations. Here is the analogous recipe:

s not divisible by	Permutation
2	(12)
6	(23)
12	(23) (45)
24	(23) (45) (67)

s is to run from 1 to 23. A brief routine to generate the recipe appropriate to the given environments is easily prepared.

Thus, canonizing a representative goes in three steps: (i) the Points are ordered according to environment; (ii) if this does not yet give an unambiguous result, the appropriate permutation recipe is generated; (iii) the permutations are applied, and the "earliest" representative according to the rule is adopted as canonical.

8. THE CONNECTED ENUMERATOR $E(y; S)$

The evaluation of the enumerator $E(y; S)$ for a given Structure S is achieved by a process akin to "inclusion-exclusion"; we now turn to consider the precise rules as developed by M. F. Sykes.

The complete enumerator is to be obtained as a sum of contributions

$$E(y; S) = \sum_{\pi} \omega_\pi E(y; S, \pi)$$

over all possible partitions π of the Points of S; ω_π is the *weight* for the partition π. Each contribution $E(y; S, \pi)$ is a product of certain polynomials, in which each Neighbor gives rise to one factor.

As an example, we examine the expression $E(y; S, AB.CD)$, where S is the Structure shown in Fig. 1. The essence of the technique is to find the partitions *induced by AB.CD* on the components of the (full) representative R of the Structure; thus

R:	$A(2)$	C	$D(2)$	$AB(2)$	BC	CD	BCD
$AB.CD$:	A	C	D	AB	$B.C$	CD	$B.CD$

The rule of formation of the *induced partitions* should be evident. As it happens, the labels AB... of the Points have now done their work, and it is sufficient henceforth to record only the "shape" of the induced partition. It is convenient to use a "decimal" notation, exemplified by

$$\underbrace{ABCD}_{1}.\underbrace{EFG.HIJ}_{2}.\underbrace{KL.MN.OP}_{3}.\underbrace{Q.R.S}_{3} \text{ has } \textit{specifier } [1233]$$

(The units place records the number of subsets with 1 member, the tens place those with 2, and so on.) The above scheme abbreviates to

R:	$A(2)$	C	$D(2)$	$AB(2)$	BC	CD	BCD
$AB.CD$	[1]	[1]	[1]	[10]	[2]	[10]	[11]

This is of course only one row of the complete table, which has a row for every possible partition of the labeled Points: see Table II. The weights depend only on the number of subsets in π, and generally take the values $\pm k!$.

Without ambiguity, we may regard the specifiers as standing for certain properly chosen *auxiliary polynomials*. This table then prescribes that, for our example,

$$\begin{aligned} E(y; S, AB.CD) &= [1]^2.[1].[1]^2.[10]^2.[2].[10].[11] \\ &= [1]^5\,[10]^3\,[2][11] \end{aligned}$$

The enumerator $E(y; S)$ is in this case the weighted sum of 15 such contributions.

Table II. The Specifier Table for the Structure of Figure 1b

	$A(2)$	C	$D(2)$	$AB(2)$	BC	CD	BCD
$\omega = 1$							
$ABCD$	[1]	[1]	[1]	[10]	[10]	[10]	[100]
$\omega = -1$							
$A.BCD$	[1]	[1]	[1]	[2]	[10]	[10]	[100]
$B.ACD$	[1]	[1]	[1]	[2]	[2]	[10]	[11]
$C.ABD$	[1]	[1]	[1]	[10]	[2]	[2]	[11]
$D.ABC$	[1]	[1]	[1]	[10]	[10]	[2]	[11]
$AB.CD$	[1]	[1]	[1]	[10]	[2]	[10]	[11]
$AC.BD$	[1]	[1]	[1]	[2]	[2]	[2]	[11]
$AD.BC$	[1]	[1]	[1]	[2]	[10]	[2]	[11]
$\omega = 2$							
$AB.C.D$	[1]	[1]	[1]	[10]	[2]	[2]	[3]
$AC.B.D$	[1]	[1]	[1]	[2]	[2]	[2]	[3]
$AD.B.C$	[1]	[1]	[1]	[2]	[2]	[2]	[3]
$BC.A.D$	[1]	[1]	[1]	[2]	[10]	[2]	[11]
$BD.A.C$	[1]	[1]	[1]	[2]	[2]	[2]	[11]
$CD.A.B$	[1]	[1]	[1]	[2]	[2]	[10]	[11]
$\omega = -6$							
$A.B.C.D$	[1]	[1]	[1]	[2]	[2]	[2]	[3]

Different applications will call for different auxiliary polynomials—and sometimes for different weights. The mere enumeration of clusters calls for

$$[1] = [10] = [100] = [1000] = \cdots = 1 + y, \ = f, \text{ say}$$
$$\text{all other auxiliary polynomials} = 1$$

We may read off from the table the 15 properly weighted terms:

$$\begin{aligned} E(y; S) &= f^{10} - 1.(f^8 + f^6 + f^7 + f^8 + f^8 + f^5 + f^6) \\ &\quad + 2.(f^7 + f^5 + f^5 + f^6 + f^5 + f^6) - 6.(f^5) \\ &= (1 + y)^5 (2y^2 + 7y^3 + 5y^4 + y^5) \end{aligned}$$

The first factor takes care of the five Neighbors which receive only one shadow; obviously these may be included or omitted in every combination—they cannot affect the connectedness of the result. The remaining factor correctly enumerates the number of ways the remaining five Neighbors may be chosen to yield a connected cluster. The $2 + 7 + 5 + 1 = 15$ ways of doing this are shown in Fig. 6.

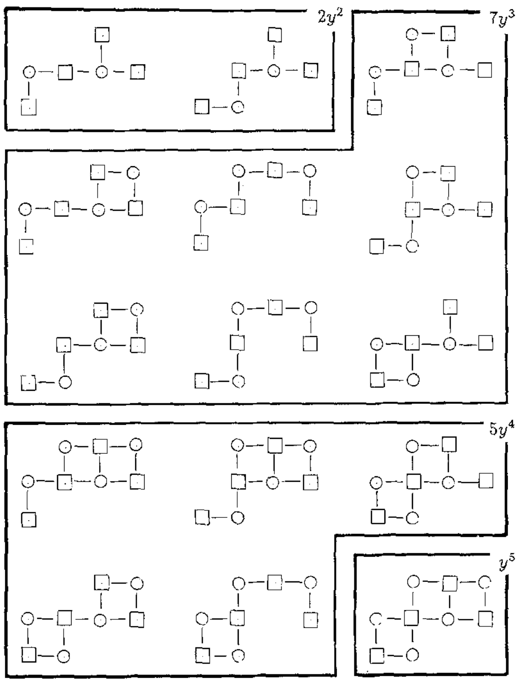

Fig. 6. The connected possibilities as enumerated by $E(y;S)$ for the X-cluster of Fig. 1.

The generation of Structures has been discussed in Sections 3 and 4, and their proper pigeonholing to yield the numbers $N(S)$ in Sections 5–7. The work of this section has completed the last link in the chain by providing a method for determining the connected enumerator $E(y;S)$ when the Structure S is known. The final assembly of the full enumerator F [Eq. (4)] is now trivial.

9. THE SCALE OF THE PROBLEM

It is important to understand that the table of specifiers for a typical Structure may be very large. The columns are not very numerous in general, but the number of rows is the total number of partitions of the labeled Points; we have seen that for four Points there are 15 rows. In general the numbers of labeled partitions, and therefore of rows, are given in Table III; they grow faster than any exponential, but not as rapidly as $m!$. [Incidentally, there is a generating function,

$$\sum_{m=0}^{\infty} \frac{a_{m+1}z^m}{m!} = e^{z-1}e^{e^z}$$

Table III. Number of Rows in a Typical Specifier Table

Number of Points (m)	Number of rows (a_m)
1	1
2	2
3	5
4	15
5	52
6	203
7	877
8	4140
9	21147
10	115975
11	678570
12	4213597

and a rather curious explicit formula,

$$a_m = e^{-1} \sum_{p=0}^{\infty} \frac{(p+1)^{m-1}}{p!} \Big]$$

The growth in number of the partition *specifiers* (for n shadows) is much more modest:

n:	1	2	3	4	5	6	7	8
Specifiers:	1	2	3	5	7	11	15	22

(66 specifiers—and therefore 66 auxiliary polynomials—in all.) For our applications there has been no need to go beyond $n = 8$: the most difficult case is the body-centered cubic lattice (of coordination $q = 8$), on which Neighbors cannot be shadowed by more than 8 Points.

For the sake of efficiency, when the number m of Points becomes large, a reference table of likely specifiers is generated as a preliminary act. There is of course one row for each labeled partition of the m Points. The number of columns is fixed by both m and the coordination q of the lattice of interest; see Table IV for the more usual values. The savings are useful for the smaller values of q, where larger values of m may well be attempted anyway. At the higher reaches the computations need to be segmented, even on the largest computers, on account of the voracious demand for storage. Segmentation according to rows of different weight ω is perhaps the neatest approach.

Table IV. Number of Columns in a Typical Specifier Table

Number of Points (m)	Number of columns for coordination q							
	$q=3$	$q=4$	$\cdots$	$q=6$	$\cdots$	8	$\cdots$	$q=\infty$
1	1	1		1		1		1
2	3	3		3		3		3
3	7	7		7		7		7
4	14	15		15		15		15
5	25	30		31		31		31
6	41	56		63		63		63
7	63	98		126		127		127
8	92	162		246		255		255
9	129	255		465		510		511
10	175	385		847		1012		1023
11	231	561		1485		1980		2047
12	298	793		2509		3796		4095

In the face of their mountainous numbers, it is futile to suppose that the necessary partitions can be compiled by hand, and a partition generator is needed. The version used in this project runs as follows.

Write a_{mr} for the number of partitions of m labeled objects into r subsets. Then

$$a_{mr} = a_{m-1,r-1} + r a_{m-1,r}$$

since the mth object may either form a new subset on its own, or else be added to one of the r already existing subsets. As is usual, the existence of such a relation suggests a procedure for the generation of the partitions themselves.

Consider the partition $A.BD.C$ (note the standardized arrangement: The items of each subset are put in alphabetic order, and then the subsets themselves are put into alphabetic order). This partition is the parent of *four* offspring in which the new object E has four different fates:

$$\begin{array}{ccccc} A.BD.C \rightarrow & AE.BD.C, & A.BDE.C, & A.BD.CE, & A.BD.C.E \\ 1232 & 12321 & 12322 & 12323 & 12324 \end{array}$$

Under each partition is written a particularly convenient code, defined recursively for each offspring by the addition of a new digit to the code of the parent. (Actually, the new digit evidently specifies where the new arrival is to be found when the new partition is written in the standard arrangement.)

The procedure for generating all the partitions is written as a procedure for generating all the corresponding codes. The rules of formation are:

(i) The partition A by itself has the code 1.

(ii) Let the maximum digit in any code be r. Then there are $r + 1$ offspring to that code, obtained by appending the digits 1, 2,..., $r + 1$ in turn.

The generation of these codes for up to four objects according to these rules, along with the corresponding partitions, is shown in Table V. The apparently confused order in the final column is not a disadvantage.

10. IS IT WORTH THE TROUBLE?

This is not the place to record the results of the project; they will be given when they come to be analyzed in other papers. However, there is no doubt that the approach described here can lead to extraordinary savings in computer time. There has to be a tradeoff, of course; the programming is very much more complicated (after all, a brute-force approach would require nothing more than Section 3 of this paper) and the demands on immediate-access computer memory are much heavier. Is it worth it?

As a simple example, consider the diamond lattice, which was used as a pilot to check the correctness of the routines. There exist 14436726016

Table V. Generating the Partitions of Four Distinguishable Objects

1	*A*	11	*AB*	111	*ABC*	1111	*ABCD*
						1112	*ABC.D*
				111	*AB.C*	1121	*ABD.C*
						1122	*AB.CD*
						1123	*AB.C.D*
		12	*A.B*	121	*AC.B*	1211	*ACD.B*
						1212	*AC.BD*
						1213	*AC.B.D*
				122	*A.BC*	1221	*AD.BC*
						1222	*A.BCD*
						1223	*A.BC.D*
				123	*A.B.C*	1231	*AD.B.C*
						1232	*A.BD.C*
						1233	*A.B.CD*
						1234	*A.B.C.D*

connected clusters of 17 sites on this lattice; to count them directly on a fast computer would take a few hours. The method of this paper requires first that 8-Point clusters on the X-lattice (a face-centered-cubic lattice, as it happens) must be generated. There are 6849415 of these; merely to count them would take a few seconds only, but the fact that they need to be pigeonholed according to Structure increases the required time substantially. Our program on the University of London CRAY X-MP took 787 sec for this task; a total of 2529 distinct realizable Structures was found. Determining the corresponding specifier polynomial (by way of a table with 4140 rows) took a further 413 sec in all; this stage of the work had to be segmented on account of the inordinate demand for computer memory. The specifier polynomial was found to comprise 8380 terms in 74 variables (66 specifiers $[\cdots]$ and 8 weights ω: the weights are kept as variables at this stage since different choices of weight may be used to solve different problems).

It remains to substitute auxiliary polynomials for the specifiers and the weights, appropriate to whichever of the several possible problems is to be solved. The most elementary substitution has been given in Section 8; it yields the numbers $c_{8,n}$ of Eq. (1) for all n. This took a further 13 sec to complete.

Thus, to carry through the work at the 8-Point level required just over 20 min of CRAY cpu time. The same procedure is naturally needed for the lower m-Point levels, $m = 2,..., 7$, and calls for another minute or so. In this way, the required count is obtained by

$$\begin{aligned} c_{17} &= c_{8,9} + c_{7,10} + c_{6,11} + c_{5,12} + c_{4,13} + c_{3,14} + c_{2,15} + c_{1,16} \\ &= 11025316374 + 3185908084 \\ &\quad + 223244454 + 2256717 + 387 + 0 + 0 + 0 \\ &= 14436726016 \end{aligned}$$

in far less time than is needed for the direct count. Worthwhile savings are thus feasible, particularly on the more challenging even lattices such as the body-centered-cubic.

Recently Styer *et al.*[8] have examined *bond percolation* in two dimensions. Part of this work involved enumerating by computer all N-bond clusters ($N = 1,..., 16$), pigeonholed by the number of perimeter bonds; the count took 27 cpu hr on the CRAY X-MP computer at the Ohio Computer Center. Such a count must include at least some 17-site connected clusters, and will be included in the information gained from a count of up to 8-Point X-clusters within the present context. This work has very recently been carried out. At the 8-Point level there are 3502 Structures, the specifier polynomial has 12271 terms, and the substitution of the appropriate auxiliary polynomials, which are much more involved in this case,[9] leads to a polynomial in three variables with 365 terms. (The final substitution generated 3706420 algebraic terms to be pigeonholed. With 1 Mword of memory dedicated to storing the pigeonholes, the garbage collector had to be invoked 22 times. These facts give a good feel for the kind of demands the method can be expected to make on memory resources.) The final outcome contains all the information needed to derive Styer's results, and much more. The cpu time ran to rather over 1000 cpu sec, also on a CRAY X-MP. Plans are in hand to extend the count to 9-Point X-clusters, and thus to obtain extensive information on 19-site clusters on the square lattice.

It is difficult to know what improvement should be claimed in this instance: so much depends on exactly what information is being gathered, and indeed on programming style. In qualitative terms, however, it is safe to say that the partnership of a fast computer and an understanding of combinatorics is likely to be unbeatable.

More can be done. For example, it is known that for certain identifiable Structures the specifier polynomial *factorizes*, and that each factor may be separately calculated by the now-familiar techniques with a further saving in time. It is likely that incorporating this fact in the procedures will serve as some defense against the unbridled growth of the numbers in Table III. How this is to be implemented is under consideration.

It is now 40 years since the appearance of Prof. Domb's early work on the pencil-and-paper methods for counting lattice embeddings. Over that time the power of the available tools has increased in ways which were far beyond the imaginations of all but perhaps the most outrageous of science fiction writers. Much has been accomplished as a result, but it would be a mistake to believe that the speed of a modern computer can now relieve us of the need to think. Remember this, and we shall be by no means at the end of the road.

ACKNOWLEDGMENTS

May I record here the pleasure and enjoyment that has been mine over the years as a result of being able to collaborate with other workers in this field, particularly with my colleagues of the Statistical Physics Group at King's College, London. I like a computing challenge; they have provided me with plenty!

REFERENCES

1. C. Domb and M. S. Green, eds., *Phase Transitions and Critical Phenomena*, Vol. 3 (Academic Press, 1974).
2. J. L. Martin, *Proc. Camb. Philos. Soc.* **58**:92–101 (1962).
3. S. Rushbrooke and J. Eve, *J. Chem. Phys.* **31**:1333 (1959).
4. M. F. Sykes, *J. Phys. A* **19**:1007–1025, 1027–1032, 2425–2429, 2431–2437 (1986).
5. J. L. Martin, in *Phase Transitions and Critical Phenomena*, Vol. 3, C. Domb and M. S. Green, eds., (Academic Press, 1974), pp. 97–112.
6. W. F. Lunnon, in *Computers in Number Theory*, A. O. L. Atkin and B. J. Birch, eds. (Academic Press, 1971); D. H. Redelmeier, *Discrete Math.* **36**:191–203 (1981).
7. D. E. Knuth, *The Art of Computer Programming*, Vol. 3 (Addison–Wesley, 1973), pp. 506–542.
8. D. F. Styer, M. D. Edwards, and E. A. Andrews, *J. Phys. A* **21**:L1153–1156 (1988).
9. M. F. Sykes, *J. Phys. A* **19**:2425–2429 (1986); Eqs. (3.2)–(3.5).