Contents

1. Introduction .............................................. 269
   1.1. On algorithms in general and lattice statistics in particular 269
   1.2. Organization of algorithms .......................... 270
   1.3. Algorithmic constructs and notation .................. 271
2. Self-avoiding walks by exact enumeration .................. 273
   2.1. The problem ........................................... 273
   2.2. The simple approach ................................... 274
   2.3. Recursion ............................................. 276
   2.4. The efficient approach ............................... 277
      2.4.1. Data organization .................................. 277
      2.4.2. Segment generation for walk counting ............. 278
      2.4.3. Walk enumeration .................................. 279
   2.5. Configurational properties ........................... 281
   2.6. Extensions and practical matters .................... 283
3. Graph generation .......................................... 284
   3.1. Background ............................................ 284
   3.2. Graph identification .................................... 285
   3.3. Star topology generation .............................. 286
      3.3.1. Overview .......................................... 286
      3.3.2. Generation of topology descendants ............... 288
      3.3.3. Edge and vertex partitioning ...................... 290
      3.3.4. How to label a topology .......................... 291
      3.3.5. Ordering the descendants .......................... 293
   3.4. Graph realizations .................................... 294
      3.4.1. Restricting and ordering realizations ............. 294
      3.4.2. Bridge length assignment .......................... 295
      3.4.3. Equivalent bridges .................................. 297
      3.4.4. Constructing the graph description ............... 298
   3.5. A direct method for graph generation .................. 299
4. Lattice constants .......................................... 299
   4.1. The problem ........................................... 299
   4.2. A simple counting algorithm ........................... 300
   4.3. The efficient approach ............................... 301
      4.3.1. Outline ............................................ 301
      4.3.2. Data organization .................................. 302
      4.3.3. Segment generation for graph counting ............. 303
      4.3.4. Revising the graph description .................... 305
      4.3.5. Fast graph counting – the initial phase .......... 309
      4.3.6. Node placement ..................................... 310
      4.3.7. Filling the bridges .................................. 312
   4.4. Correcting for symmetry ............................... 314
5. Graph weights for series expansion ........................................... 315
   5.1. Introduction ...................................................................... 315
   5.2. Data representation ............................................................ 316
   5.3. Weight computation ............................................................ 316
      5.3.1. Subgraph generation ..................................................... 316
      5.3.2. Expansion of the weight function ................................. 318
6. Self-avoiding walks – the Monte Carlo approach .......................... 320
   6.1. The problem ...................................................................... 320
   6.2. Outline of method .............................................................. 320
   6.3. Data organization .............................................................. 322
   6.4. Walk generation from fragments ......................................... 323
   6.5. Walk properties ............................................................... 326
7. Percolation by Monte Carlo ....................................................... 328
   7.1. Background ...................................................................... 328
   7.2. Techniques for cluster construction ..................................... 328
      7.2.1. The basic method ....................................................... 328
      7.2.2. Multiple labeling ....................................................... 330
      7.2.3. Slab subdivision ........................................................ 331
   7.3. Slab generation algorithm .................................................. 332
      7.3.1. Overall structure ........................................................ 332
      7.3.2. Row processing ........................................................ 333
      7.3.3. Label compression ...................................................... 334
      7.3.4. Postprocessing .......................................................... 336
   7.4. Assembling the lattice ....................................................... 336
      7.4.1. Organizational details .................................................. 336
      7.4.2. Assembly algorithm .................................................... 337
      7.4.3. Putting clusters together .............................................. 339
      7.4.4. The remaining computations ........................................ 341
8. Monte Carlo algorithm for the discrete spin model ......................... 341
   8.1. Background ................................................................. 341
   8.2. The model and its representation ....................................... 341
   8.3. Pretabulation ................................................................. 344
   8.4. The algorithm ............................................................... 346
   8.5. Properties and extensions .................................................. 347
9. Afterthoughts ........................................................................ 348
References ........................................................................ 349
ALGORITHMS FOR LATTICE STATISTICS

D.C. RAPAPORT

Department of Physics, Bar-Ilan University, Ramat-Gan, Israel

Received 25 June 1987

Computers play an ever-increasing role in modern science. The goal of effective algorithm design is to achieve greater benefit from this resource which, surprisingly or otherwise, is never available in sufficient quantity. In this tutorial paper we examine a number of different types of algorithm that have been used in computations on lattice statistical systems. These case studies in algorithm design are the following: 1) An exact enumeration study of the self-avoiding walk. 2) The problem of generating collections of graphs. 3) The evaluation of lattice constants. 4) The computation of graph weights for use in series expansions. 5) An alternative approach to the self-avoiding walk using a Monte Carlo technique. 6) A Monte Carlo study of the percolation problem. 7) A new vector spin model treated by Monte Carlo simulation. In several of these problems the enhanced capability resulting from a careful approach to the design of the algorithm made it possible to extend the range of the computations beyond what had previously been considered feasible. An awareness of how particularly heavy computational tasks can be approached, including the benefits accruing from appropriately organizing the data used in the computations, ought to contribute to the more effective use of computers for these and similar types of problem in the future.
1. Introduction

1.1. On algorithms in general and lattice statistics in particular

How better to begin than with a definition: An algorithm is a ‘systematic procedure that produces – in a finite number of steps – the answer to a question or the solution to a problem’ [1]. This all-embracing definition can be further elaborated upon to satisfy the purists [2]. On the other hand it is questionable whether the word itself should exist at all – see ‘algorism’ in ref. [3].

Lattice statistics is a term that covers the statistical mechanics of interacting systems that are confined to lattices; among the more popular systems are spin models typified by the Ising and Heisenberg models, polymer models based on the self-avoiding walk, cluster growth problems such as percolation, and lattice gas models of fluids. In some circumstances the use of a lattice is justified by the underlying reality – the spins of a ferromagnet are after all located on the sites of a crystal lattice; in other instances the lattice is introduced in an ad hoc fashion in the hope of forcing the problem into a more tractable form – fluids and polymer chains do not really reside on lattices under normal conditions. In the quest for complete analytic solutions the lattice has not proved to be of great help, with the notable exception of the two-dimensional Ising model and its immediate relations. But when one turns to numerical methods there is little doubt that the simplification resulting from the use of a lattice is significant. A host of numerical techniques have been called into play – series expansions, Monte Carlo, renormalization group, phenomenological renormalization, cellular automata (a recent addition) to name but some. A great deal has been learned about the model systems and the knowledge transferred to realistic situations (the laboratory and the real world), but as the more accessible properties of these systems are measured innate curiosity leads one onwards in two directions: towards improved resolution and to the exploration of properties that are less easily measured. This results in further extensive numerical studies in a cycle which shows little sign of abating; despite the fact that many of the models have been in circulation for decades there seems to be a continuing growth in the interest they attract.

The fact that computational algorithms of various kinds play a key role in lattice statistics might lead to the reasonable assumption that the techniques are well documented in the literature, in a manner comparable to, say, the standard algorithms of numerical analysis, e.g. ref. [4]. Unfortunately this is far from true. The absence of such information presents a problem for the reader who is interested in reproducing or extending the results, assessing their reliability (and possibility validity), or adapting the techniques to other problems. This absence may be related to the question of whether it is considered intellectually respectable to publish computational techniques [5]; however the lack of a means to evaluate published work is not consistent with established practice in the much older approaches to scientific endeavor – analytic theory and experiment – where it is taken for granted that derivations and procedures must be described in verifiable detail.

In an effort to begin to fill the void, and in the hope of encouraging others to do likewise, we have compiled a collection of algorithms developed and used in a variety of typical lattice-statistical studies. The techniques were outlined only briefly in the original publications that reported the results, and this is the first time that they are presented in detail. The intention of this article is not, however, merely to serve as a collection of recipes, although it may indeed fill this
function; it is also a tutorial on the design and implementation of computational algorithms addressed primarily to lattice statisticians. But, since the problems themselves are so simple to state, although not to solve, this fact should not be allowed to deter others.

The algorithms are tailored to specific problems; in the quest for efficiency – an important issue since the calculations can consume a lot of computer time – there is little room to accommodate too much generality. It is unlikely that the algorithms applied to these particular problems can produce substantial further progress until computer speeds (and availability) improve significantly. The intention is that at least some of the algorithms described should be regarded as case studies: how a certain basic approach is modified to produce an efficient computational scheme, and the role played by the careful organization of the data used in the computation. These principles, together with selected techniques (suitably cannibalized), should prove of value in treating other problems of a related nature.

The particular problems and methods discussed are the following: The self-avoiding walk is treated from two points of view – as a problem for exact enumeration and as the subject of a Monte Carlo study. The three distinct elements of the computations involved in producing series expansions for spin models are described; these are the generation of a set of lattice graphs, the counting of graph embeddings to produce lattice constants and a particular instance of the computation of graph weights – the example being the susceptibility of the Ising model. A Monte Carlo technique for studying the percolation problem is described, with emphasis on the computation of the distribution of cluster sizes. The final case study involves the so-called discrete spin model, a hybrid of the Ising and Heisenberg models which retains many of the characteristics of the latter but at the same time approaches the former in computational simplicity. The selection is a personal one; inclusion of everybody's favorite algorithm would probably necessitate an entire book. The results obtained by the various methods are not described here since they have already been detailed elsewhere; reference to them will be made as required, as well as to reviews and background material of a more general nature.

Algorithms must coexist with and operate on data. The more data that an algorithm is responsible for dealing with, the more important in many situations the effort that goes into organizing the data. Simple one-dimensional tables and two-dimensional arrays (or vectors and matrices) provide the most basic form of data-organization; data arranged with a view to optimal algorithmic performance can involve more complex structures using, for example, pointers linking data items that are to be accessed according to a sequence differing from the order in which they are stored. Other examples of data structuring will be encountered later on. In extreme cases, once the data have been suitably structured the algorithmic prescription becomes almost trivial. It is the case that in a number of the algorithms presented here the data organization is a critical factor in the overall efficiency.

1.2. Organization of algorithms

The order in which the algorithms are presented is not completely arbitrary. Enumerating self-avoiding walks is in essence a simplified case of the more general graph counting problem; what is learned in the simpler problem can be carried over to the more complex one and the presentation of the latter shortened to some extent. The three sections on graph generation, lattice constants and graph weights describe the computations needed for series expansions in the
order the problems are dealt with in practice. Similarly, the Monte Carlo approach to the walk problem builds on experience gained in dealing with exact enumeration. The sequence of presentation is an approximate guide to the order in which the material ought to be read, rather than by random access. It should be added that in order to prevent the length of this article getting out of hand some of the work is left for the reader; details of a number of the less edifying and entirely straightforward algorithmic sequences are left as exercises for the interested reader (who is unlikely to complete them unless the algorithm is to be rendered operational).

One question which arose while preparing the article was whether to include actual program listings. Since the ideal programming language has yet to be created, rather than be fettered by arbitrary syntactic limitations we have chosen to use a homebrewed dialect that, though derived from existing computer languages, is freely adaptable to the author’s whim. The rules will be defined shortly; no difficulty in producing an operational program is any one of a variety of languages is envisaged. This translation should also prove to be an easier task than converting an algorithm already coded in one language into another, an oft-performed activity. Since the algorithms are not intended to be viewed as ‘black boxes’, and to use the algorithm for anything other than the problems for which they were originally designed requires making changes, which in turn implies an understanding (hopefully) of the underlying logic, the translation to the language of choice can be treated as a familiarization process.

A number of well-established programming practices are avoided in the interest of clarity. It is often the case that a program can be made to run faster by replacing multiply-dimensioned arrays by vectors and doing the index computations explicitly, more than one quantity can be packed into a single computer word (this practice is retained in some of the algorithms where it is an important contributor to efficiency), and temporary variables can be used to avoid repeated index computations. These and other ‘tricks’ can be included when converting an algorithm into a program, but are inappropriate when the goal is a didactic one. Suggestions as to where these enhancements might be introduced are included with some of the algorithms; in certain instances they are all too obvious. Needless to say, the improvements resulting from such refinements are very strongly dependent on both the computer hardware and the optimization ability of the language compiler used – in some situations there may be little or no gain whatsoever.

Finally, several shortcuts have been taken in the interests of brevity, clarity and because inclusion of all the details would tend to introduce language and computer system dependencies. There is no explicit specification of data types or array sizes; these will be obvious from the context in which the variables appear. There are no checks for illegal input data, arrays exceeding bounds, and so on; these are examples of good programming practice and should always be included, but not when trying to convey the essence of the workings of an algorithm. No details are given in regard to input and output processes, problems associated with handling data stored in disk files, and how checkpointing can be used to break a long computation into a series of shorter steps. Ditto for program testing and performance evaluation. These are assumed to be subjects with which the implementer is (or is about to become) well versed.

1.3. Algorithmic constructs and notation

A prerequisite for understanding the algorithms is a familiarity with the algorithmic constructs used as well as a number of common and uncommon shorthand notations. The richer the
notation the more concise the description of the algorithm, but this will not be pushed to the
limit (cf., a programming language such as APL).

Block structuring is used with indentation to enhance the logical organization. All variables
are of global scope in the sense that they can be accessed from any point in the algorithm. The
much maligned 'goto' operation is used only when its avoidance by artificial means would make
the algorithm even harder to follow; the destination of the 'goto' is embedded in slashes. The
block constructs are almost self-explanatory:

if condition1 then
...
elseif condition2 then
...
else
...
endif,

while condition do
...
endwhile,

repeat
...
until condition,

for variable = low_range to high_range & condition do
...
endfor.

The test for 'while' is performed prior to each iteration, as is the test for 'for', whereas that for
'repeat' is carried out after each iteration; at least one iteration of the 'repeat' loop is always
carried out, but it is possible for the 'while' and 'for' loops to be skipped if the entry conditions
are not met. There is an optional condition test included with the 'for' which determines which
of the loop iterations are carried out. A simple single line

if condition then...

statement is also provided. The 'skip' operation bypasses the remainder of the current loop
iteration; 'break' terminates the loop immediately. A procedure mass with 'procedure name
(arguments):', ends with 'endproc', can be exited at some intermediate point by means of a
'return', and is invoked by specifying the name alone. An 'exit' terminates the algorithm at some
intermediate point.

The usual arithmetic operations are employed, with ** denoting exponentiation. Where single
bit (equivalent to Boolean) operations are appropriate for the variables concerned, the operators
\( \lor \), \( \land \) and \( \oplus \) correspond, respectively, to the Boolean 'or', 'exclusive or' and 'and'. \( \ldots = \{ \ldots \} \)
denotes a test for the equality of two unordered sets. The operated ++ and -- are shorthand
for increment and decrement by unity. Logical values ('true' or 'false' – the results of
comparisons) can be compounded using the operators & and | which denote 'and' and 'or'. The
reverse arrow ← denotes assignment; ↔ denotes an exchange of values. An asterisk indicates that the operation is carried out for the entire range of meaningful values of that index. Multiple levels of indexing are allowed. The ' ' is used to make the names of variables more intelligible (contrary to the practice enforced by certain programming languages, Fortran in particular, we prefer to give longer names to variables in order that their function be easier to comprehend), and '#' is often appended to names of variables that are used for indexing or counting purposes. Single statements too long to be included on a single line use an ellipsis to indicate their continuation to a subsequent line.

2. Self-avoiding walks by exact enumeration

2.1. The problem

The self-avoiding walk, or SAW, is a model of a freely linked polymer chain [6]. Because the SAW is free to wander unhindered (by external influences) over configuration space there is the implicit assumption that the chain is in dilute solution free from the interference of other chains; the model is not intended to apply to polymers that adopt fixed spatial structures such as the proteins.

The only interaction that appears in the model is that due to excluded volume – no two monomers can occupy the same region of space simultaneously; this is the way in which the SAW differs from the readily solved random walk model [7] and is responsible for the fact that the SAW problem has not yielded to analytic methods. Embedding the walk in a lattice is done for convenience and in the entirely reasonable hope that the properties are not changed beyond recognition; excluded volume becomes the restriction that the walk may not visit the same lattice site more than once.

The questions which tend to be asked most often relate to the asymptotic behavior; to cite one example, how does the mean-square end-to-end distance $\langle R_N^2 \rangle$ vary with the number of steps $N$ in the limit $N \to \infty$? All the evidence points to a result of the form $\langle R_N^2 \rangle \sim N^{2v}$, and the problem then becomes one of determining the exponent $v$ to a satisfactory degree of accuracy. Once this is known it becomes feasible to study the correction terms to this asymptotic result, the higher order moments, the full distribution of end-to-end distances, and so on.

The exact enumeration method for treating the SAW problem forms the subject of this section. In a subsequent section the alternative Monte Carlo technique is described. While the statistical Monte Carlo approach is capable of generating samples of walks that are long enough to be practically in the asymptotic regime, those produced by exact enumeration have to be counted in their entirety, and thus the lengths achievable are far removed from the onset of asymptotic behavior. Extrapolation techniques [8] must be applied to the exact count data in order to project the results in the direction of the asymptotic limit and these introduce an element of uncertainty since assumptions are necessary concerning the functional dependence on $N$. Monte Carlo results on the other hand are subject to statistical uncertainty. Results of recent calculations of both kinds are to be found in ref. [10] and references therein.

Efficient algorithms are essential in order to extend the walk lengths that can be reached by exact enumeration. One such algorithm is presented later in this section. By way of introduction,
this efficient, but somewhat intricate algorithm is preceded by a very simple version that is also perfectly capable of counting SAWs, although it takes longer to complete the task.

Exact enumeration is also beset by a more fundamental problem, namely that for a lattice of coordination number \( q \) the amount of work to add each additional step grows by a factor of almost \((q - 1)\); in the case of the face-centred cubic lattice this amounts to an order of magnitude each time. Thus there is a more fundamental limit to growth, with algorithmic efficiency providing only limited relief.

There are no lack of extensions to the SAW model, and the family of related models continues to grow. One variant aims to incorporate the effects of the solvent; depending on the mutual affinity of monomer and solvent the chain can expand or collapse as the temperature is lowered. The simplest model capable of reproducing this effect does not represent the solvent explicitly but instead introduces interactions between monomers that happens to occupy adjacent lattice sites in a particular conformation. Another member of the enlarged SAW family is the trail, a walk that is prevented from traversing a lattice edge more than once. A brief discussion of how the algorithm might be adapted to problems of this type is included.

2.2. The simple approach

The simplest approach to the problem of counting self-avoiding walks is the direct one. It is a simple backtracking algorithm, in the sense that a series of systematic attempts are made to place the next step of the walk on the lattice but if the algorithm finds all possible routes blocked it retreats one step, moves that step to a different lattice edge if possible and then advances forward again to the next step. The initial step of the walk begins at a fixed lattice site “origin”. Such an approach will produce all SAWs of a specified length if the algorithm is allowed to run to completion.

A more complete description of the process just outlined is the following; the variables introduced in the body of the algorithm are defined immediately afterwards, although the meaning of many, if not all, should be apparent from the context.

Initialization –
- site_flag(*) ← empty
- walk_count ← 0
- site_loc(0) ← origin
- site_flat(origin) ← full
- step# ← 0

Add single step to walk –
/1/  ++ step#
- index(step#) ← 0

Attempt to position latest step on a lattice edge –
/2/  repeat
    if index(step#) = lattice_degree then goto /3/
    ++ index(step#)
    site_loc(step#) ← site_loc(step# - 1) + lattice_vector(index(step#))
until site_flag(site_loc(step#)) = empty
Test whether walk still incomplete –
if step# < walk_length then
Flag lattice site reached by step (this is not the final step) –
site_flag(site_loc(step#)) ← full
  goto /1/
else
Walk is complete (there is no need to flag final site) –
  ++ walk_count
  goto /2/
endif

Remove latest step and clear site reached by previous step (algorithm terminates when back at (start) –
/3/  – step#
  if step# > 0 then
    site_flag(site_loc(step#)) ← empty
    goto /2/
  endif

Only a minimal investment of effort would be required to convert this algorithm into a working program. The variables used in the algorithm are described as follows, listed in order of appearance:

- **site_flag** – An array of flags, one per lattice site, showing whether the sites have been visited by the current walk (either “full” (=1) or “empty” (=0)).
- **walk_count** – A running count of the complete walks generated so far.
- **site_loc** – The sequence of site locations visited by a particular walk.
- **step#** – Step counter.
- **index** – Used to count the number of attempts to place each step; counting is restarted whenever the step is removed during backtracking.
- **lattice_degree** – The number of nearest neighbors (coordination number) of a lattice site.
- **lattice_vector** – Displacements of the nearest neighbor sites (see below).
- **walk_length** – The required number of steps.

The lattice itself may be two- or three-dimensional (or any other for that matter), but for convenience it is mapped onto the set of integers. Thus an \(L \times L \times L\) simple cubic lattice maps onto the integers 1 through \(L^3\), with “origin” located in the middle. The mapping must preserve the structure and connectivity of the lattice; thus if \(r_3 = r_1 + r_2\), where the \(r_i\) denote vectors between lattice sites, then if \(\mu(r)\) is the mapping function to the integers, it must satisfy \(\mu(r_3) = \mu(r_1) + \mu(r_2)\). The mapping itself implies periodic boundaries, and \(L\) must be large enough to prevent even the longest walk from returning to the origin after completing a circuit of the periodic lattice. While the lattice mapping can be carried out in various ways, the simplest is the one in which the offsets to the six neighbouring lattice sites, and hence the values of “lattice_vector” used in the algorithm, are \(\pm 1, \pm L, \pm L^2\).
Before going on to the development of a more efficient algorithm for SAW counting we digress briefly to show how the use of recursion can produce a more concise and elegant description of the algorithm just given.

2.3. Recursion

Recursion [11] allows the algorithm just described to be expressed in simpler form.

```
site_flag(\ast) \leftarrow \text{empty}
walk_count \leftarrow 0
site_flat(origin) \leftarrow \text{full}
next_step (1, origin)

procedure next_step(step#, previous_site):
for index# = 1 to lattice_degree do
    new_site \leftarrow previous_site + lattice_vector(index#)
    if site_flag(new_site) = empty then
        if step# = walk_length then
            ++ walk_count
        else
            site_flag(new_site) \leftarrow \text{full}
            next_step(step# + 1, new_site)
        endif
    endif
endfor
endproc
```

The procedure “next_step” keeps calling itself in a recursive fashion until all steps of the walk are in place; if at any stage a blockage is encountered the procedure exits to a previous incarnation in order to shift an earlier step. When the original call to the procedure terminates the counting process is complete. Recursion simplifies the algorithm by removing the need to maintain arrays of quantities associated with the individual steps; it does this by automatically generating new local copies of the quantities concerned each time the procedure calls itself. It is clear from the context which of the variables are used in this manner—“index#”, “previous_site”, “new_site” and “step#”. The other variables are of global scope, in the sense that a single copy is common to all the nested calls to the procedure.

Despite the fact that a recursive point of view is ideally suited to a number of the algorithms described here, it is not the approach of choice for one overriding reason. Recursion is not supported by all computer language implementations, and even where it is, the design of the processor hardware does not allow it to be carried out in an efficient manner. Each successive call to the procedure entails a usually considerable overheat that cannot be tolerated in a lengthy computation where such calls are made very frequently; in the case of the SAW algorithm a call is made every time the final step is changed which is, indeed, the most repeatedly performed
piece of the computation. Thus, since the emphasis here is on practical recipes rather than
elegant representational forms, the recursive approach must be deferred until such time as it can
be as efficiently implemented as the non-recursive.

2.4. The efficient approach

2.4.1. Data organization

An approach that is more effective from the computational point of view involves building the
walks, not just one step at a time, but by adding several steps simultaneously. The sequences of
steps to be added – short SAWs hereinafter referred to as segments – are generated in advance;
in fact a table of self-avoiding segments, suitably organized to allow it to be used in an efficient
manner, is generated prior to beginning the task of counting walks. Two separate algorithms are
thus required, one to produce the table of segments, the other to count the walks. The advantage
of the approach will become obvious when the algorithm is described in detail.

The segments are generated in order of increasing length up to a specified limiting length
(typically 4–6, depending on lattice). The order in which the segments appear corresponds to a
generation scheme in which the final step varies the most frequently, the penultimate step less
frequently, and so on back to the initial step. Since some or all of the steps prior to the last will
in general be replicated over a series of several segments appearing consecutively in the segment
table, it is possible to devise a compact mode of storing the segment data that avoids the need for
repeated copies of the earlier steps; such an approach not only conserves space but contributes to
the efficiency of the walk counting process.

The simplest possible segment table would consist merely of sets of walk steps, a fixed number
of steps for each segment of a given length (this is what is used in the Monte Carlo SAW
algorithm – see later). The compact version associates a linkage pointer with each step stored;
this link points from one step in a segment to the corresponding position in the next segment
listed in the table which experiences a change at this particular step, but whose earlier steps
remain the same. By construction, all segments between these two (if there are any) have all steps
up to and including the one in question unchanged, and it is precisely this redundant informa-
tion that can be omitted. If no such subsequent segment exists, in other words the next change in
this step coincides with a change in an even earlier step, the stored pointer value is set to zero
and the SAW counting algorithm, whose principal activity consists of traversing the segment
table, must take the appropriate action to find the earlier step which has to be altered. The idea
of preparing the SAW segments in advance was originally proposed in ref. [12] in connection
with graph counting; we will also use the segment approach in our treatment of this subject
(later), although the generation method will be changed to produce an ordering more suited to
the problem.

The segment tables are stored as a pair of one-dimensional arrays (vectors), the first
“segment_step” contains the actual lattice displacements of sites from their predecessors (the
starting site is not mentioned since the displacements are relative), the second “segment_link”
either points to the location of the next segment where the particular step experiences its next
change or has the value zero to indicate that backtracking to an earlier step of the segment is
necessary. For convenience several initial entries in the link vector are set aside to serve as
pointers to the first entry in the table for each segment length. Most segment entries will contain
just a single step – the final one; this fact becomes increasingly true as the lattice coordination number is increased. Other information can be stored together with the segments, in particular data that can be used in computing the configurational properties of the walks (see later).

2.4.2. Segment generation for walk counting

The algorithm to create the segment table is based on the SAW counting method described earlier. The table is generated as a step preparatory to the walk counting. New variables that are used are defined following the algorithm.

Initialization –

\begin{align*}
\text{site}._\text{flag}(\ast) & \leftarrow \text{empty} \\
\text{segment}._\text{pointer} & \leftarrow \text{max}._\text{length} \\
\text{site}._\text{loc}(0) & \leftarrow \text{origin} \\
\text{site}._\text{flag}(\text{origin}) & \leftarrow \text{full}
\end{align*}

Loop over range of segment lengths –

\begin{align*}
\text{for length} \# & = 1 \text{ to max}._\text{length} \text{ do} \\
\text{changed}._\text{step} & \leftarrow 1 \\
\text{previous}._\text{pointer}(1) & \leftarrow \text{length} \# \\
\text{step} \# & \leftarrow 0
\end{align*}

Add step –

\begin{align*}
&/1/ \quad ++ \text{step} \# \\
&\quad \text{index(\text{step} \#)} \leftarrow 0
\end{align*}

Attempt to position latest step on lattice edge –

\begin{align*}
&/2/ \quad \text{repeat} \\
&\quad \text{if index(\text{step} \#) = lattice}._\text{degree} \text{ then goto } /3/ \\
&\quad \quad ++ \text{index(\text{step} \#)} \\
&\quad \quad \text{site}._\text{loc(\text{step} \#)} \leftarrow \text{site}._\text{loc(\text{step} \# - 1)} + \text{lattice}._\text{vector} \left(\text{index(\text{step} \#)}\right)
\end{align*}

until site._flag(site._loc(\text{step} \#)) = \text{empty}

Test whether segment still incomplete –

\begin{align*}
&\quad \text{if \text{step} \# < \text{length} \# \text{ then} } \\
&\quad \quad \text{site}._\text{flag(site}._\text{loc(\text{step} \#))} \leftarrow \text{full} \\
&\quad \quad \text{goto } /1/ \\
&\quad \text{endif}
\end{align*}

Segment complete; set pending pointer (belonging to a previous segment table entry) –

\[
\text{segment}._\text{link(previous}._\text{pointer(\text{changed}._\text{step}))} \leftarrow \text{segment}._\text{pointer} + 1
\]

Save steps of complete segment in table; all new pointers are tentatively set to zero –

\begin{align*}
&\text{step} \# \leftarrow \text{changed}._\text{step} + 1 \\
&\text{repeat} \\
&\quad ++ \text{step} \# \\
&\quad ++ \text{segment}._\text{pointer} \\
&\quad \text{segment}._\text{step(segment}._\text{pointer}) \leftarrow \text{lattice}._\text{vector} \left(\text{index(\text{step} \#)}\right) \\
&\quad \text{segment}._\text{link(segment}._\text{pointer}) \leftarrow 0 \\
&\quad \text{previous}._\text{pointer(\text{step} \#)} \leftarrow \text{segment}._\text{pointer} \\
&\quad \text{until \text{step} \# = \text{length} \#}
\end{align*}
changed_step ← length#
goto /2/
Remove latest step –
/3/  -- step#
if step# > 0 then
    site_flag(site_loc(step#)) ← empty
    changed_step ← step#
    goto /2/
endif
endfor

Variables here for the first time are as follows (again in order of appearance):

segment_pointer  – Latest entry in segment table.
max_length  – Maximum segment length required.
length#  – Current segment length.
changed_step  – The earliest step in the segment to change; used in deciding how many
                of the steps need to be stored in the segment table.
previous_pointer  – Gives location of a pending pointer – one whose value can only be set in
                    the light of subsequent information.
segment_link  – Pointers for traversing segment table (described in text); the initial
                 “max_length” words are pointers to the first table entries for each
                 segment length.
segment_step  – Actual lattice displacements for steps (described in text): the initial
                 “max_length” words are unused.

2.4.3. Walk enumeration
In order to be able to use the segment data just described, the SAW itself must be represented
as a sequence of segments, in other words divided into several sequences of steps each of length
“max_length”, with one possible additional sequence containing fewer steps. For want of a more
suitable term we will call each of these step sequences a bridge. The single shorter bridge, if any,
is used to start the walk. The enumeration problem has now become one of filling the bridges
with segments of the correct length in all possible ways, subject to the self-avoidance condition.
The algorithm which follows does just this.

Initialization –
    site_flag( · ) ← empty
    walk_count ← 0
    site_loc(0) → origin
    site_flag(origin) ← full
    bridge# ← 0
Add bridge and set pointer to initial segment table entry for corresponding length –
/1/  ++ bridge#
    step# ← step_total(bridge# − 1) + 1
index(step#) ← bridge_length(bridge#)
go to /3/

Select next segment from table if available –
/2/ index (step#) ← segment_link(index(step#))
if index(step#) = 0 then goto /4/

Attempt to position steps of segment on lattice edges –
/3/ site_loc(step#) ← site_loc(step# - 1) + segment_step(index(step#))
if site_flag(site_loc(step#)) ≠ empty then goto /2/
if step# < step_total(bridge#) then
    ++ step#
    index(step#) ← index(step# - 1) + 1
    goto /3/
endif

test whether walk still incomplete –
if bridge# < walk_ bridges then
flag lattices sites occupied by latest bridge –
    for step# = step_total(bridge# - 1) + 1 to step_total(bridge#) do
        site_flag(site_loc(step#)) ← full
    endfor
    goto /1/
endif

Walk complete –
    ++ walk_count
    save_properties

Remove single step of bridge (if any remain) –
/4/ -- step#
    if step# > step_total(bridge# - 1) then goto /2/

Remove bridge (since none of its steps remains) and test if all bridges are gone
(end of algorithm) –
    -- bridge#
    if bridge# > 0 then
        Clear sites occupied by previous bridge –
            for step# = step_total(bridge# - 1) + 1 to step_total(bridge#) do
                site_flag(site_loc(step#)) ← empty
            endfor
            goto /2/
        endif
    endif

The new variables introduced are the following:

bridge# – Bridge counter.
step_total – The total number of steps in the bridges up to and including the one referenced (indexing begins at zero).
index – Usage of this array has been modified; each element now serves as a pointer to the appropriate step location within the segment table.
It is clear from a study of this algorithm that the site occupancy flags for the sites on which
the final segment resides are never set (or cleared) since the segments themselves are self-avoid-
ing walks there is absolutely nothing to be gained by such an operation. This represents a major
saving in execution time since it reduces considerably the amount of work that has to be done in
the most frequently executed portion to the algorithm. The penalty which has been paid to
achieve this saving is that the overall algorithm has grown in length and complexity, and that
storage for the segment table must be made available. The fact that the segments themselves are
self-avoiding produces a further reduction in computing effort.

A considerable amount of unnecessary walk generation can be eliminated if the symmetry of
the lattice is taken into account In the case of the fcc (face-centred cubic) lattice the first step of
the walk can be placed on one of twelve equivalent edges leading away from the origin; clearly it
suffices to choose just one of them and then multiply the final count by twelve. The second step
can proceed in four non-equivalent directions out of a total of eleven. The third step represents
the limit for which a gain in speed is significant; here only 32 of 44 possible directions (for all the
distinct second steps) need to be considered. Thus the first three bridges of the walk should be
assigned unit length; the counting algorithm should be applied to the bridges formed by
grouping the remaining steps and the resulting walk counts multiplied by the corresponding
symmetry factors (there are 32 of these for the fcc lattice and their computation is left as an
exercise for the curious reader). The gain is most marked for the fcc lattice, but is significant for
all lattices.

2.5. Configurational properties

Two of the principal properties of self-avoiding walks are the mean-square end-to-end
distance and radius of gyration, defined, respectively, as

\[ R_N^2 = \langle (r_N - r_0)^2 \rangle, \]

\[ S_N^2 = \frac{1}{N+1} \left\langle \sum_{i=0}^{N} (r_i - \bar{r}_v)^2 \right\rangle, \]

where

\[ \bar{r}_v = \frac{1}{N+1} \sum_{i=0}^{N} r_i. \]

The walk in question has N steps, starting at \( r_0 \) and visiting sites \( r_0, \ldots, r_N \); the average \( \langle \ldots \rangle \) is
over all possible walk realizations (weighted with the symmetry factors) generated by the
counting algorithm. Much of the computation involved in evaluating \( R_N^2 \) and \( S_N^2 \) can be
eliminated by storing certain data about distances together with the segment table, and storing
yet further data in a table associated with the actual lattice sites. We discuss each of these in turn.

Assume the walk to be divided into $K$ segments; the length of the $k$th segment is $n_k$ and obviously

$$
\sum_{k=1}^{K} n_k = N. \tag{4}
$$

Now the position of the $i$th site visited – which is actually the $l$th site of the $k$th segment – can be expressed as

$$r_l = g_k + h_{kl}, \tag{5}
$$

where $g_k$ is the position of the lattice site on which the segment begins and $h_{kl}$ is the relative displacement of the $l$th site of this segment with respect to its initial site. The decomposition (5) allows (2) to be rewritten as

$$S_N^2 = \frac{1}{N+1} \langle \sigma_1 \rangle + \frac{1}{(N+1)^2} \langle \sigma_2^2 \rangle. \tag{6}
$$

where

$$
\sigma_1 = \sum_{i=0}^{N} r_i^2 = \sum_{k=1}^{K} n_k g_k^2 + 2 \sum_{k=1}^{K} g_k \cdot \alpha_k + \sum_{k=1}^{K} \beta_k \tag{7}
$$

$$
\sigma_2 = \sum_{i=0}^{N} r_i = \sum_{k=1}^{K} n_k g_k + \sum_{k=1}^{K} \alpha_k. \tag{8}
$$

It is assumed that $r_0 = 0$. The quantities $\alpha_k$ and $\beta_k$ are functions of the distances between sites visited by individual segments and can therefore be computed at the same time the segment table is generated; they are defined as

$$
\alpha_k = \sum_{l=1}^{n_k} h_{kl}, \tag{9}
$$

$$
\beta_k = \sum_{l=1}^{n_k} h_{kl}^2. \tag{10}
$$

The values of $\alpha_k$ and $\beta_k$, once suitably tabulated, can be accessed through the same indices used for determining the segment displacements.

The quantities $g_k$ and $g_k^2$ can also be obtained from precomputed values stored in a table, in this case one indexed by the lattice site number. For the simple cubic lattice each site number $l$ can be uniquely expressed as a sum of the displacements associated with the lattice vectors, namely $l = l_x + L l_y + L^2 l_z$, where $l_x$, $l_y$, and $l_z$ are integers in the range $(-L, +L)$. The square
of the distance to the site from the origin \((l_x = l_y = l_z = 0)\), i.e. \(g_k^2\), is simply \(l_x^2 + l_y^2 + l_z^2\), assuming unit distance between neighboring sites. Similarly, \(g_k\) has components \(l_x, l_y, l_z\). These values are computed in advance and stored in such a way that the lattice site number \(I\) provides immediate access to them. Note however that in order to conform with the need for each site to have a unique distance from the origin (the periodic boundary conditions implied by the lattice mapping can result in multiple values for the distance depending on the route taken) the maximum walk length cannot exceed \(L/2\); the lattice must have twice the minimum linear dimension required just for counting walks if distances are also to be computed.

To give another example of a lattice description consider the fcc lattice. It has 12 possible displacements to nearest neighbors; for a lattice consisting of \(L^3\) unit cells, and hence \(4L^3\) sites, these displacements are \((\pm 1, L, L^2 - L - 1, L^2 - L, L^2 - 1, L^2)\). The origin of each unit cell is given by \((L - 1)l_x + (L + 1)l_y + (2L^2 - L - 1)l_z\), where \(l_x, l_y\) and \(l_z\) are integers, and the offsets within the cell involve an additional \(Lm_x, m_y, m_z\) \((L^2 - 1)m_x, m_y, L^2m_z\), where any two of \(m_x, m_y, m_z\) are unity and the third zero. The squared distance to the site is then \(\sum (2l_x + m_x)^2 + (2l_y + m_y)^2 + (2l_z + m_z)^2\). Other lattices can be similarly treated.

Computation of \(R^2_N\) and \(S^2_N\) has been reduced to the summation of quantities that are precomputed and easily retrieved. This represents yet another example of the way computing speed (still a relatively expensive resource if needed in great quantity) can be gained at the cost of increased storage (a cheaper resource).

The walk-counting algorithm presented in this section is readily extended to include computation of \(R^2_N\) and \(S^2_N\); in fact the extension allows the computation of the complete distributions of these quantities, from which even-order moments beyond the second can be evaluated. The sums \(\sum n_k g_k^2\) and \(\sum n_k g_k\) must be modified each time the operations of adding and removing a bridge are encountered; to simplify this work intermediate cumulative sums can be generated and indexed by “bridge#” so that recalculation is only required when a bridge is added. The remaining sums must be modified immediately prior to the test for a complete walk. The value of the squared end-to-end distance and radius of gyration (the latter multiplied by a factor \((N + 1)^2\)) for each walk that is completed successfully are used to generate a histogram whose bin width is unity; since the maxima of (1) and (2) are simple functions of \(N\), namely \(N^2\) and \(N(N + 2)/12\), respectively, the upper limits to the histogram ranges are known. Further details are left to the reader.

### 2.6. Extensions and practical matters

One straightforward extension of the preceding algorithm is the problem of lattice trails. Whereas the SAW is characterized by not being allowed repeated visits to any lattice site, the trail is subject to a similar restriction being placed on the lattice edges. To adapt the algorithm for trail counting requires a systematic numbering of lattice edges in order to allow checking for repeated edge usage. To reduce computation the segment table can be augmented with information concerning the (relative) edge number of each step to help determine the actual edges involved as the segment is placed on the lattice. Results of a calculation of this type appear in [13].

Self-interacting walks are another extension of the model that present an additional difficulty. In this model, as was pointed out earlier, an interaction is present if two (nonconsecutive) sites
visited by the walk are adjacent. Thus, once a walk has been generated it is necessary to count the number of adjacent site pairs. The count of such pairs is best accumulated as the walk grows, and again the segment table can be used to hold data that reduces the amount of calculation, in this instance the number of nearest neighbor site pairs within each segment. A possible further enhancement would entail the creation of an array, one value per lattice site, giving the potential numbers of neighbors that a future bridge would encounter based on considering the steps of the walk already in place; such an array would not need to be modified by the final bridge and would eliminate the need to count site pairs associated with this bridge. Results for this model (which are overdue for extension) are to be found in ref. [14].

Self-avoiding rings [15] are yet another member of the SAW family; these are walks that start and terminate at the same location, but are otherwise self-avoiding. The procedure for handling this calculation resembles that of the SAW to some extent, but it is in fact a particular case of graph counting described in a subsequent section.

We conclude this discussion with a brief mention of a few points regarding implementation of the algorithm. All the computations involve integer arithmetic only – there are no floating-point numbers involved; it is only when the weighted averages are computed from the histograms in evaluating $R_N^2$ and $S_N^2$ at the end of the entire computation that a little floating-point arithmetic is needed. The flag that indicates whether a lattice site is occupied or not takes only two values (typically 1 and 0) and could be represented by a single bit; however arithmetic performed on selected bits of computer words is generally a relatively expensive process in comparison with regular word-length operations and it is more effective to assign a larger storage element (of word length or similar) for each lattice flag even though this increases the storage required. Finally, all the quantities associated with a segment – the steps, pointers and distances – can be stored as part of just a single vector. The pointer arithmetic will need to be altered to enable the correct quantities to be located at each stage of the calculation, but the fact that only one vector is used rather than several (six in the case of the $S_N^2$ calculation) can lead to more efficient code on many kinds of computers.

3. Graph generation

3.1. Background

Graphs, of one kind or another, have proved to be extremely useful as mnemonics for the individual terms of perturbation expansions. Lattice statistics and graph theory have had a long and flourishing relationship; review of theory and applications are to be found in ref. [9]. The three distinct steps involved in producing a perturbation expansion, or series expansion to use the terminology of the lattice statistician, are (i) the generation of a list of graphs of the type demanded by the problem, (ii) the computation of the number of ways each can be embedded in the lattice (the lattice constants), (iii) the evaluation of the contribution of each graph to the series (the weights). Different kinds of expansion are required for different classes of problem; we will restrict ourselves to one particular type of problem – the high-temperature expansion for the Ising model. The graph generation process is discussed in this section; how to go about computing lattice constants and graph weights is covered in subsequent sections. We will assume
that the reader has some familiarity with this subject; namely the ways in which graphs are used in the expansions of lattice statistics, the types of graphs involved in problems of interest, and the kinds of information that can be extracted from the series once generated. For a more general perspective on graph enumeration see ref. [16].

Graph theory has a rich vocabulary; an extensive compilation of the terminology will be found in ref. [17]. Put simply, a graph is a set of edges that meet at vertices; for our purposes there are no isolated vertices, nor are there loops—edges that have both ends incident on the same vertex, nor do edges have directions associated with them. The vertices of the graph may or may not be labeled. The graphs may be connected, or consist of two or more connected components. A multiply-connected graph allows at least two fully independent ways of traversing the edges between any pair of vertices; such a graph is called a star for convenience. An automorphism is a one-to-one correspondence between the vertices which results in a similar correspondence between the edges, in other words, an embedding of the graph—both vertices and edges—in itself. The automorphisms form a group, the order of this group is the symmetry number of the graph. Two graphs are isomorphic if one can be shown to correspond to an automorphism of the other. The degree of a vertex is the number of edges incident on it; vertices with degree greater than two are called nodes. A bridge is a sequence of edges (joining degree-two vertices) between a pair of nodes. Graphs which differ from one another only in respect to the number of vertices of degree two are said to be homeomorphic, and a graph with no such vertices is homeomorphically irreducible, or for brevity, if it is also multiply-connected, a star topology (or just plain topology). We define the realization of a topology to be the graph obtained when zero or more degree-two vertices are inserted into the edges of the topology. The cyclomatic number of a graph is \( c = e - v + 1 \) where \( e \) is the number of edges and \( v \) the number of vertices; \( c \) is invariant under homeomorphism. Finally, our use of the term embedding implies a weak embedding (for the distinction see ref. [17]), and is a way of identifying the vertices of a graph \( g \) with those of another (generally larger) graph \( G \), that ensures all the edges of \( g \) are also edges of \( G \). Graph \( g \) is then said to be a subgraph of \( G \); the number of ways the embedding can be carried out is a quantity that is often required, but if \( G \) is an infinite lattice the embeddings are carried out with one arbitrarily chosen site of \( g \) maintained at a fixed site of the lattice.

We begin with a discussion of the practical issue of identifying graphs, and then proceed to describe two approaches to the generation of collections of graphs. The first approach involves the steps: the production of topologies, and the subsequent derivation of their realizations. The alternative approach, one of generating the graphs directly, is discussed only briefly. The choice of method (or possibly some combination of the two) is open, but the problem under consideration often suggests which is the more suitable, in the sense that most of the effort goes into producing just those graphs that are actually needed by the problem.

### 3.2. Graph identification

In order to facilitate the automated processing of graph descriptions a means of representing the graph in the computer is required. This is actually the simple part of a more complicated problem. The difficult part is to ensure that the description of the graph is a unique one; ambiguities will lead to the same graph appearing any number of times under different guises. The problem is equivalent to that of determining whether a pair of graphs is isomorphic. There is
no elegant general way of answering this question that falls significantly short of an exhaustive comparison of the ways in which the graphs are constructed; certain easily obtained characteristics which might at first sight be taken to be unique for each graph (to within an isomorphism) turn out to allow ambiguity once the graphs become sufficiently complicated. Since the labeling issue is central to the graph generating algorithm we will briefly describe the techniques used to deal with the problem. More on the subject is to be found in refs. [18–20].

There are many ways of describing a graph. In the course of the algorithms which follow the same graph may appear with different, but equivalent descriptions; alternative kinds of description are best suited to the various stages of processing. A graph can be specified by giving a list of the vertex pairs on which edges are incident. It can also be expressed in terms of its adjacency matrix $A = (a_{ij})$, where $a_{ij}$ gives the number of edges joining vertices $i$ and $j$. Yet another means of description is a list of the node pairs joined by bridges together with a similarly ordered list of the bridge lengths.

The problem now is to determine a unique way of identifying a graph, irrespective of the actual mode of description, so that graphs that are isomorphic will have the same identity. A simple method that is guaranteed to work is to start with vertices arbitrarily labeled $1, \ldots, v$ and to write down the elements of the corresponding adjacency matrix in linear order (row-by-row) forming a number $B$. If the graphs do not have multiple edges this can be regarded as a single binary number. Then permute the vertex labels in all possible ways until $B$ is maximized; the resulting arrangement of the vertex labels can be declared to be the canonical one, and $B$ (or some equivalent quantity) the identifier of the graph. Graphs sharing the same (canonical) identifier are isomorphic. A graph dictionary (for a set of graphs obeying certain criteria) could then be constructed by, for example, listing the graphs in order of decreasing $B$; there will be no duplicates in such a dictionary.

The solution just proposed is simple but problematic: since there are $v!$ permutations of the labels of a $v$-vertex graph (with perhaps some reduction if graph symmetry is allowed for) the computation necessary to maximize $B$ grows very rapidly as a function of $v$. There are ways to reduce the number of label permutations examined based on constraining the permissible assignments of labels to vertices. The most obvious constraint is to require that the labels are ordered by vertex degree; thus $i < j$ implies that the degree of vertex $i$ is not less than that of $j$. This scheme helps, but does not solve the problem for graphs with sufficiently many (or all) vertices of the same degree. Further possibilities include the classification of vertices according to the degrees of the vertices attached to them, or to count the number of walks (including those that self-intersect) of selected lengths along the graph edges that return to each vertex. For the latter scheme, the number of $n$-step walks returning to vertex $i$ is simply the $i$th diagonal element of $A^n$. Even this does not always provide the answer; the polygon is an easily detected graph where these methods are of little assistance, but there are others that are not so readily identified in advance. Thus the approach to graph identification is perforce an heuristic one, but one that is adequate for the majority of graph sizes encountered so far.

3.3. Star topology generation

3.3.1. Overview

Star topologies – hereinafter just topologies for brevity – are (as defined earlier) multiply-con-
connected graphs with no vertices of degree less than three (all vertices are nodes) and which are permitted to have multiple edges between vertices. Topologies with \(e\) edges and \(v\) vertices (where \(e > v + 1\)) are generated by adding edges and vertices (i.e., nodes) to previously produced topologies of certain sizes. Three sets of topologies of different sizes contribute to the desired set of \((e, v)\) topologies –

(i) \((e - 1, v)\) to which a new edge is added by joining a pair of existing vertices;
(ii) \((e - 2, v - 1)\) to which a new vertex is added by insertion into an existing edge and joined using a new edge to an existing vertex;
(iii) \((e - 3, v - 2)\) to which two new vertices are added into either the same edge or two separate edges and are then joined with a new edge.

None of the three processes can generate the entire \((e, v)\) set on its own, and many of the resultant topologies will be generated several times. The three sets of topologies that are input can be regarded as parents, the topologies output by the generation process as descendants. The one unifying feature of the three processes is that the cyclomatic number increases by unity, in other words the descendants have one more independent cycle than their parents.

Much of the redundancy inherent in the generation technique can be eliminated by taking into account the symmetries of the parent topologies, although some replication (due to descendants having multiple parents) will remain that can only be eliminated by detailed comparison of the descendants. For the purpose of this discussion, vertices which map to each other under automorphism are regarded as equivalent; the same holds true for edges, and multiple edges between a pair of vertices are mutually equivalent. Then the number of descendants produced by each of the processes is reduced in the following way –

- process (i): if vertices \(v_1\) and \(v_2\) are equivalent there is no need to consider new edges incident at \(v_2\), except for the edge joining \(v_1\) to \(v_2\);
- process (ii): if edges \(e_1\) and \(e_2\) are equivalent the new vertex need not be inserted into \(e_2\);
- process (iii): if \(e_1\) and \(e_2\) are equivalent there is no need to add the two new vertices to \(e_2\), nor is there any need to insert a single vertex into \(e_2\) to be joined to a new vertex in another edge unless this is \(e_1\).

This line of argument could be pushed further to include, for example, combinations of equivalent edges and/or vertices, but the resulting savings are unlikely to justify the effort involved.

There is only one topology with cyclomatic number \(c = 1\), namely the polygon. The so-called theta graph with \(e = 3, v = 2\) is the only \(c = 2\) topology. For \(c > 2\) the possible combinations of \(e\) and \(v\) start to grow, as do the patterns of connectivity that characterize distinct topologies. The set of \((e, v)\) topologies is obtained by subjecting complete sets of types \((e - 1, v)\) \((e - 2, v - 1)\) and \((e - 3, v - 2)\) to the appropriate processes described above. The set of descendants of each parent is sorted during generation to eliminate remaining duplicates, then merged into the set of descendants of previously processed parent topologies, again with erasure of duplicate copies. While for a given \(v\) there is no upper limit to \(c\) in principle, for practical purposes those topologies for which \(e\) exceeds \(v\) by too large an amount do not contribute to the expansions for which the graphs are destined since their realizations can no longer be embedded in the lattice. Topologies are therefore normally generated in order of increasing \(v\), with a suitable upper limit being set for the corresponding \(e\) (or \(c\)).

The algorithm involved in the generation process is described in the following subsections.
3.3.2. Generation of topology descendants

The algorithm produces a set of descendants using any one of the processes just described. To eliminate a certain amount of tedious repetition we assume, in the description of this algorithm only, that each operation applied to the adjacency matrix is repeated in suitably modified form to preserve its symmetry. Furthermore we omit operations that restore the vertex degrees and adjacency matrix elements following the production of each new descendant; these can be incorporated with little effort.

Construct adjacency matrix –

\[
\text{adj\_matrix}(\ast, \ast) \leftarrow 0 \\
\text{degree}(\ast) \leftarrow 0 \\
\text{for edge}\# = 1 \text{ to } \text{graph\_edges} \text{ do} \\
\quad + \quad \text{degree(\text{start\_vertex(edge}\#))} \\
\quad + \quad \text{degree(\text{end\_vertex(edge}\#))} \\
\quad + \quad \text{adj\_matrix(\text{start\_vertex(edge}\#), \text{end\_vertex(edge}\#))} \\
\text{endfor}
\]

Partition edges and vertices into sets equivalent under automorphism –

determine\_equivalences

Generate descendants using process (i) –

if new\_vertex = graph\_vertices then

\[
\text{for vertex1}\# = 1 \text{ to } \text{graph\_vertices} - 1 \text{ & equiv\_vertex(vertex1}\#) = 0 \text{ do} \\
\quad + \quad \text{degree(vertex1}\#) \\
\text{for vertex2}\# = \text{vertex1}\# + 1 \text{ to } \text{graph\_vertices} \text{ do} \\
\quad + \quad \text{degree(vertex2}\#) \\
\quad + \quad \text{adj\_matrix(vertex1}\#, \text{vertex2}\#) \\
\quad \text{label\_canonically} \\
\quad \text{merge\_topology} \\
\text{endfor}
\]

endfor

Generate descendants using process (ii) –

elseif new\_vertex = graph\_vertices + 1 then

\[
\text{degree(new\_vertex)} \leftarrow 3 \\
\text{for edge}\# = 1 \text{ to } \text{graph\_edges} \text{ & equiv\_edge(edge}\#) = 0 \text{ do} \\
\quad - \quad \text{adj\_matrix(\text{start\_vertex(edge}\#), \text{end\_vertex(edge}\#))} \\
\text{adj\_matrix(\text{start\_vertex(edge}\#), \text{new\_vertex}) \leftarrow 1} \\
\text{adj\_matrix(\text{new\_vertex}, \text{end\_vertex(edge}\#)) \leftarrow 1} \\
\text{for vertex}\# = 1 \text{ to } \text{graph\_vertices} \text{ do} \\
\quad + \quad \text{degree(vertex}\#) \\
\quad + \quad \text{adj\_matrix(vertex}\#, \text{new\_vertex}) \\
\quad \text{label\_canonically} \\
\quad \text{merge\_topology} \\
\text{endfor}
\]

endfor
Generate descendants using process (iii) –

`elseif new_vertex = graph_vertices + 2 then`

`degree(new_vertex - 1) ← 3`
`degree(new_vertex) ← 3`

`for edge1 # = 1 to graph_edges & equiv_edge(edge1 #) = 0 do`
`    adj_matrix(start_vertex(edge1 #), end_vertex(edge1 #)) ← 0`

Both vertices on same edge –

`    adj_matrix(start_vertex(edge1 #), new_vertex - 1) ← 1`
`    adj_matrix(new_vertex - 1, new_vertex) ← 2`
`    adj_matrix(new_vertex, end_vertex(edge1 #)) ← 1`
`    label_canonically`
`    merge_topology`

Vertices on separate edges –

`    adj_matrix(new_vertex - 1, new_vertex) ← 1`
`    for edge2 # = edge1 # + 1 to graph_edges do`
`        adj_matrix(start_vertex(edge2 #), end_vertex(edge2 #)) ← 0`
`        adj_matrix(start_vertex(edge2 #), new_vertex) ← 1`
`        adj_matrix(new_vertex, end_vertex(edge2 #)) ← 1`
`        label_canonically`
`        merge_topology`
`    endfor`

`endif`

Variables used in this portion of the generation algorithm, excluding obvious counting variables, are:

- `adj_matrix` – The adjacency matrix of the parent topology suitably altered to represent the descendant.
- `degree` – Vertex degrees of parent/descendant.
- `graph_edges` – Number of edges in parent topology.
- `start_vertex` – Initial vertex on which edge is incident.
- `end_vertex` – Terminal vertex on which edge is incident.
- `new_vertex` – Number of vertices specified for descendant topology.
- `graph_vertices` – Number of vertices in parent.
- `equiv_vertex` – Indication of whether vertex is first member of equivalence partition (≠ 0), or not (≠ 0).
- `equiv_edge` – Ditto for edge.

The procedures “determine equivalences”, “label canonically” and “merge topology” do precisely what the names imply; they are discussed below. In order to generate the required set of topologies the above algorithm must be applied in turn to all (as many as three) sets of potential parent topologies.
3.3.3. Edge and vertex partitioning

To reduce the number of topology replications the generation algorithm assumes that edges and vertices have been partitioned into equivalence classes. The following algorithm achieves this partitioning by first testing for multiple edges between vertices, and then examining (by means of a backtracking approach) edges and vertices which map onto one another under all possible automorphism transformations.

```
procedure determine_equivalences:
  equiv_vertex(•) ← 0
  equiv_edge(•) ← 0

Examine multiple edges (these appear consecutively) –
  for edge# = 2 to graph_edges do
    if \{start_vertex(edge#), end_vertex(edge#)\} = \{start_vertex(edge# - 1), end_vertex(edge# - 1)\} then equiv_edge(edge#) ← 1
  endfor

Generate mappings of graph onto itself (i.e., embeddings) –
  vertex_flag(•) ← empty
  vertex# ← 1
  vertex_loc(1) ← 1

Add vertex to embedded graph –
/1/ vertex_flag(vertex_loc(vertex#)) ← full
    ++ vertex#

Look for vacant vertex location with same degree –
  vertex_loc(vertex#) ← 1
/2/ if degree(vertex#) = degree(vertex_loc(vertex#)) & vertex_flag(vertex_loc(vertex#)) = empty then
    Check for correct neighbor vertices –
      vertex2# ← vertex# - 1
      while vertex2# > 0 do
        if adj_matrix(vertex#, vertex2#) ≠ adj_matrix(vertex_loc(vertex#), vertex_loc(vertex2#)) then goto /3/
        vertex2# ← vertex2#
      endwhile
      Test for completed embedding –
        if vertex# < graph_vertices then goto /1/

Mark equivalent vertices and edges –
  for vertex# = 1 to graph_vertices & equiv_vertex(vertex#) = 0 do
    if vertex_loc(vertex#) < vertex# then equiv_vertex(vertex#) ← 1
  endfor
  for edge1# = 1 to graph_edges - 1 ∈ equiv_edge(edge1#) = 0 do
    for edge2# = edge1# + 1 to graph_edge & equiv_edge(edge2#) = 0 do
      if \{vertex_loc(start_vertex(edge2#)), vertex_loc(end_vertex(edge2#))\} = ...
```
Variables not met previously are as follows:

vertex_flag — Used to indicate whether a vertex has another mapped onto it.
vertex_loc — Gives identity of vertex onto which the particular vertex is mapped.

Note that only vertex equivalences are required by topology generation process (i), while (ii) and (iii) only refer to the edge equivalences; one could therefore be selective as to which equivalences are examined, although the additional effort is minimal.

3.3.4. How to label a topology

The canonical identifier of each generated topology is produced by following the principles discussed earlier; it is subsequently used in the preparation of an ordered list of distinct topologies. The identifier, corresponding to the value of B when the vertices are labeled in the prescribed sequence, is the output of the following procedure.

procedure label_canonically:
   Order vertices (highest degree first) and group into blocks according to degree –
   sort_key(∗) ← degree(∗)
   total_blocks ← 0
   vertex_order(0) ← 0
   degree(0) ← 0
   for vertex# = 1 to graph_vertices do
      biggest ← 1
      for vertex2# = 1 to graph_vertices do
         if sort_key(vertex2#) > sort_key(biggest) then biggest ← vertex2#
         vertex_flag(vertex_loc(vertex#)) ← empty
         vertex_loc(vertex#) ← vertex_loc(biggest) + 1
         vertex_order(0) ← vertex_order(0) + 1
         vertex_flag(vertex_loc(vertex#)) ← empty
      endfor
      vertex_loc(vertex#) ← vertex_loc(biggest) + 1
      vertex_order(0) ← vertex_order(0) + 1
   endfor
   for vertex# = 1 to graph_vertices do
      if vertex# > 1 then
         -- vertex#
         vertex_flag(vertex_loc(vertex#)) ← empty
         goto /3/
      endif
   endfor
endproc
endfor
vertex_order(vertex#) ← biggest
sort_key(biggest) ← 0
if degree(vertex_order(vertex#)) = degree(vertex_order(vertex# - 1)) then
   ++ block_size(total_blocks)
else
   ++ total_blocks
   block_start(total_blocks) ← vertex#
   block_size(total_blocks) ← 1
endif
perm_pointer(vertex#) ← block_start(total_blocks) - 1
endfor

Run through all permutations (see below) of vertices within blocks to ‘maximize’ adjacency matrix –
   can_order(*) ← vertex_order(*)
/1/ for block# = 1 to total_blocks & block_size(block#) > 1 do
Permute vertices within block –
   swap_index ← block_start(block#) + 1
even# ← 0
/2/ ++ perm_pointer(swap_index)
Check if this vertex exchange leads to next permutation –
   if perm_pointer(swap_index) < swap_index then
      if even# ≠ 0 then
         swap_index2 ← block_start(block#)
      else
         swap_index2 ← perm_pointer(swap_index)
      endif
      vertex_order(swap_index) ↔ vertex_order(swap_index2)
   endif
Examine change in adjacency matrix after vertex permutation –
   for vertex1# = 1 to graph_vertices - 1 do
      for vertex2# = vertex1# + 1 to graph_vertices do
         change ← adj_matrix(vertex_order(vertex1#), vertex_order(vertex2#)) ...
         - adj_matrix(can_order(vertex1#), can_order(vertex2#))
         if change ≠ 0 then
            If adjacency matrix ‘increased’ retain current vertex permutation –
               if change > 0 then can_order(*) ← vertex_order(*)
               goto /1/
         endif
      endfor
   endfor
   goto /1/
endif
Additional work to get next permutation (if all have not been generated already) –
   perm_pointer(swap_index) ← block_start(block#) - 1
++ swap_index
    even# ← 1-even#
    if swap_index < block_start(block#) + block_length(block#) then goto /2/
endfor
endproc

Variables:

sort_key – The key on which the sort is made, namely the vertex degrees.
total_blocks – Number of blocks of vertices with equal degree.
vertex_order – Rearranged sequence of vertices.
biggest – The highest degree vertex found.
block_size – Number of vertices in block.
block_start – First vertex (after degree ordering) in block.
perm_pointer – Pointer used to keep track of permutation.
can_order – Rearranged vertex sequence corresponding to tentative canonical labeling, and to actual canonical labeling when procedure terminates.
swap_index – One of the quantities (i.e., vertex labels) to be exchanged.
even# – Also helps in keeping track of the permutation.
swap_index2 – The other quantity to be exchanged.
change – Difference in adjacency matrix elements due to permutation.

Permutation within blocks is achieved by exchanging pairs of vertices; this particular permutation algorithm (there are many) is described in ref. [21]. The sorting technique is the simplest possible; others are to be found in ref. [22] but the size of the set to be sorted is unlikely to warrant a more elaborate approach.

Since there are \( k! \) permutations of \( k \) vertices in a block the possibility of having an excessive number of vertices with the same degree must be guarded against. A further criterion for classifying vertices (described above), that will in general reduce the block sizes is to consider the number of two- and perhaps three-step returns to each vertex, values readily obtained from the diagonals of the adjacency matrix raised to the appropriate power. The initialization of the vector “sort_key” would then use values derived from a combination of these walk counts together with the vertex degrees, instead of the degrees alone.

3.3.5. Ordering the descendants

The canonically labeled set of vertices is passed to the procedure “merge_topology” whose responsibility is the merging of the new topology with those produced previously, ensuring that it is either placed in the correct position in the list or discarded if it proves to be a duplicate. Prior to this, however, the topology description should be converted to a compact form that can be manipulated efficiently.

A compact representation of the topology, one that become increasingly effective as the cyclomatic number grows, is based on packing the upper triangle of the adjacency matrix (excluding the all-zero diagonal) into as few data words as possible. If, for example, no vertex is of degree higher than seven, just three bits will be sufficient to hold any element of the matrix.
Ten matrix elements can therefore be compressed into the usual 32-bit computer word. The number of words required to store the compacted matrix is a simple function of the number of vertices. The procedure below carries out the desired packing, allowing three bits per matrix element.

```
procedure pack_matrix:
    word# ← 0
    field# ← 0
    for vertex1# = 1 to new_vertex - 1 do
        for vertex2# = vertex1# + 1 to new_vertex do
            if field# = 0 then
                field# ← max_fields
                1 + word#
                packed_matrix(word#) ← 0
            endif
            −− field#
            packed_matrix(word#) ← packed_matrix(word#)
            + adj_matrix(can_order(vertex1#), can_order(vertex2#)) * 8 ** field#
        endfor
    endfor
endproc
```

After "pack_matrix" has provided "merge_topology" with the compressed adjacency matrix the next step of the processing is to insert this block of data into the already stored list of data blocks for previous descendants of the topology. This amounts to no more than a simple search through an ordered table, followed by a shift of the latter portion of the table to make room for the new data which can be copied into the vacated space; if the data is identical to an item already in the table it is simply discarded. Finally, when all descendants of a particular parent (or several parents) have been generated, the ordered list is merged into a cumulative list of descendants of parent topologies already treated – only a single pass through the two sets of data suffices for the merge since both are already correctly ordered. Expansion of the packed matrices to recover the edge description of the topologies is also a straightforward operation (a related computation is described further on). These aspects of the processing are left for the reader.

3.4. Graph realizations

3.4.1. Restricting and ordering realizations

The star topologies just discussed lack vertices of degree two. For certain problems it is necessary to take the topologies generated and insert additional (degree-two) vertices to produce star graphs with a prescribed number of edges but which remain homeomorphic to the original topologies. A systematic method of producing such graphs from the topologies is described in this section. In order to make the distinction between topologies and realizations more apparent
in the subsequent discussion, the edges and vertices of the former will be referred to as bridges and nodes from now on.

If the graphs are destined for use with a particular lattice then certain characteristics of the lattice can be used to eliminate entire classes of graphs from the generation process. A graph of cyclomatic number \( c \) contains \( c \) independent cycles; the minimum size of each cycle is dictated by the lattice — for close-packed lattices it is three, for loose-packed four, and for the diamond and hexagonal six. In addition to this limitation, a loose-packed lattice only admits cycles of even length. Other graphs can be eliminated because the maximum node degree is bounded by that of the lattice. If the topology has multiple bridges between pairs of nodes then, in the realization, only one such bridge can have unit length. There is also the obvious limitation that the least number of edges in a realization cannot be less than the sum of the minimum bridge lengths of the topology from which it is derived.

In producing realizations the problem of replication again arises. The algorithm must use information concerning multiple bridges and bridge equivalences under automorphism (of the topology) to eliminate such redundancy. The canonical vertex labeling of the topology itself leads to a unique ordering of the bridges when the adjacency matrix is expanded. If this is supplemented by a list of the bridge lengths ordered in the same way, a complete description of the graph realization is obtained. An order relation for the set of bridge lengths can be defined; \( \{ d_1, d_2, \ldots \} \) is said to precede \( \{ d'_1, d'_2, \ldots \} \) if \( d_1 < d'_1 \), or \( d_1 = d'_1 \) and \( d_2 < d'_2 \), etc. Provided only those realizations whose bridge sets precede all others of equivalent (in the automorphic sense) realizations of the topology are included there will be no problem of replication.

3.4.2. Bridge length assignment

Minimum assigned bridge lengths are determined based on the above criteria. Restrictions arising from specific lattices are not included here but can be easily added. Replication is avoided by referring to a table of equivalent bridges whose construction is detailed in the next subsection.

```plaintext
Assign minimum bridge lengths –
start_node(0) ← 0
end_node(0) ← 0
spare_edges ← graph_edges - graph_bridges
for bridge# = 1 to graph_bridges do
  if { start_node(bridge#), end_node(bridge#) } = { start_node(bridge# - 1), end_node(bridge# - 1) } then
    min_length(bridge#) ← 2
    -- spare_edges
  else
    min_length(bridge#) ← 1
  endif
endfor
Check if required number of graph edge reached –
if spare_edges = 0 then
```

build realization
exit
endif

Build table of bridge equivalences under automorphism -
equivalent bridges

Assign bridge lengths in all possible non-equivalent ways -
length(*) ← min_length(*)
length(1) ← length(1) + spare_edges
bridge_assign(*) ← 1

Check that multiple bridges are in non-decreasing length order -
/1/ for bridge# = 2 to graph bridges & min_length(bridge#) > 1 do
  if length(bridge#) < length(bridge# - 1) then goto /2/
endfor

Check that bridge length ordering is minimal -
equiv# ← 0
while equiv# < equiv_total do
  bridge# ← 0
  repeat
    ++ bridge#
    if length(bridge#) > length(equiv_bridge(equiv# + bridge#))
      then goto /2/
  until bridge# = graph bridges
  equiv# ← equiv# + graph bridges
endwhile

build realization

Next bridge length assignment -
/2/ if bridge_assign(1) < graph bridges then
  -- length(bridge_assign(1))
  ++ bridge_assign(1)
  ++ length(bridge_assign(1))
  goto /1/
else
  edge# ← 1
  while edge# < spare_edges do
    ++ edge#
    if bridge_assign(edge#) < graph bridges then
      length(graph bridges) ← min_length(graph bridges)
      -- length(bridge_assign(edge#))
      ++ bridge_assign(edge#)
      length(bridge_assign(edge#)) ← length(bridge_assign(edge#)) + edge#
      edge2# ← edge# − 1
      repeat
        bridge_assign(edge2#) ← bridge_assign(edge#)
This is another example of a backtracking algorithm; it differs from its predecessors in that the edges (or equivalently, the vertices) that are being inserted into the bridges do not have distinct identities, unlike previous examples where the quantities concerned (graph vertices, self-avoiding walk segments) were all distinguishable.

Of the variables used in the algorithm a few are renamed versions of variables appearing in the treatment of topologies: “start_node”, “end_node” and “graph_bridges” replace “start_vertex”, “end_vertex” and “graph_edges” for describing the topology. The remaining variables are as follows:

- spare_edges: The number of additional edges to be allocated among the bridges.
- graph_edges: The number of edges in the realization.
- min_length: Minimum number of edges in the bridge.
- length: Current bridge length.
- bridge_assign: A means of keeping track of which bridges contain the extra edges.
- equiv#: Offset to entry in equivalent-bridge table.
- equiv_total: Total number of automorphisms recorded in equivalent-bridge table.
- equiv_bridge: Table of bridge equivalences under the automorphisms of the topology.

### 3.4.3. Equivalent bridges

Multiple copies of realizations are avoided by referring to a table of bridges that can be brought into equivalence under an automorphism of the topology. In this context several bridges incident on the same pair of nodes are considered mutually equivalent. The method is almost the same as that used for mapping the graph onto itself when partitioning edges and vertices during topology generation. In addition to a few changes to variable names (reflecting the switch to nodes is place of vertices for the topologies) the processing differs once an automorphisms is found; only this portion of the procedure is detailed.

```
procedure equivalent_bridges:
    equiv_total ← 0
    automorph# ← 0
     Generate an embedding of the graph into itself –
     (see “determine_equivalence”)
     Store corresponding bridges unless this is first automorphism (the identity) –
     ++ automorph#
    if automorph# > 1 then
        for bridge# = 1 to graph_bridges do
```
if min_length(bridge1#) = 1 then
  mult_bridge# ← 0
Find earliest bridge equivalent to this bridge –
  bridge2# ← 0
repeat
  ++ bridge2#
  until \{ node_loc(start_node(bridge2#)), node_loc(end_node(bridge2#))\} = \ldots
  \{ start_node(bridge1#), end_node(bridge1#)\}
  equiv_bridge(equiv_total + bridge2#) ← bridge1#
else
A set of multiple bridges are equivalent to the first such bridge –
  ++ mult_bridge#
  equiv_bridge(equiv_total + bridge2# + mult_bridge#) ← bridge1#
endif
endfor
equiv_total ← equiv_total + graph_bridges
endif
Proceed to next embedding –
(see "determine equivalences")

3.4.4. Constructing the graph description
This is a simple exercise in developing the edge lists, starting from the topology description together with the bridge lengths.

procedure build_realization:
vertex# ← 0
edge# ← 0
vertex_label(*) ← 0
for bridge# = 1 to graph_bridges do
  ++ edge#
  if vertex_label(start_node(bridge#)) = 0 then
    ++ vertex#
    vertex_label(start_node(bridgeyn)) ← vertex#
  endif
  start_vertex(edge#) ← vertex_label(start_node(bridge#))
while length(bridge#) > 1 do
  ++ vertex#
  end_vertex(edge#) ← vertex#
  −− length(bridge#)
  ++ edge#
  start_vertex(edge#) ← vertex#
endwhile
if vertex_label(end_node(bridge#)) = 0 then
  ++ vertex#
3.5. A direct method for graph generation

The method described above involves a two-stage process, topology production followed by
generation of the realizations. The alternative is to generate the actual graphs directly. While
only star graphs were produced by the preceding method, the version of the direct method to be
outlined produces all connected graphs (without multiple edges or loops).

Graphs with \( e \) edges and \( u \) vertices (where \( e \geq u \)) are generated from previously produced
\((e - 1, v)\) graphs by adding a single edge which connects a pair of vertices not already sharing an
edge. Beginning with the set of \( v \)-vertex Cayley trees \((v - 1, v)\), which are connected graphs
having no cycles, edges are added to produce sets of \((e, v)\) graphs for increasing \( e \) up to the
maximum possible of \( v(v - 1)/2 \). The trees \((v - 1, v)\) are themselves generated from \((v - 2, v -
1)\) trees by adding an edge that is incident on only a single vertex. The algorithm bears a strong
resemblance to that used for the topologies and a minimal description should suffice; further
details are to be found in ref. [23].

In order to avoid excessive replication of descendant graphs the parent graph automorphisms
are again used to limit the number of different attempts to add a new edge. Canonical labeling is
carried out as previously, although the increased likelihood of encountering graphs with many
vertices having the same degree (the graphs involved tend to have more vertices than the
topologies) will almost certainly require that the proposed extension to the vertex ordering rules
be incorporated. Since the adjacency matrix elements for graphs without multiple edges are zero
or one only, a single bit per matrix element is sufficient for storage. In other respects the
generation process is similar to that of the topologies.

4. Lattice constants

4.1. The problem

The discussion in this section focuses on how to actually evaluate weak lattice constants,
principally those for star graphs. The problem amounts to obtaining an answer to the question:
given a graph defined in terms of vertices and edges, in how many distinct ways can this graph
be embedded in a regular lattice? Since the lattice is infinite the problem is actually one of
finding the number of ways per lattice site. Furthermore, since the graph is unlabeled, any
symmetry of the graph will result in overcounting; the definition of the lattice constant requires
that the count be corrected through division by the symmetry number. We begin the discussion
with a simple counting algorithm which makes up in clarity what it lacks in efficiency.
4.2. A simple counting algorithm

The simple approach uses the edge list of the graph directly (and is not limited to counting stars). There is a certain subtle order that must be present in the edge list of the graph which the counting algorithm described below relies upon. The ordering requirement is that when a new vertex appears in the edge list of the graph, all other edges (if any) that join this vertex to previously encountered vertices immediately follow in the edge list. The way in which this assumption is used in the algorithm will become apparent in due course. The algorithm for converting an arbitrarily (e.g., manually) produced graph description into one that is correctly ordered is a simple one; it will not be included at this stage but a closely related algorithm is discussed later on. The graph description constructed by "graph_realization" in the previous section does not satisfy this ordering requirement, but is in fact oriented towards a visual presentation of the graph.

Considerable effort is saved by the introduction of the array "site_access" which is used in the decision whether a pair of lattice sites are nearest neighbors and can thus be joined by an edge of the embedded graph; this array is indexed by the offset (measured in terms of site numbers – see discussion earlier in connection with self-avoiding walks) between sites and must be large enough to include all possible offsets that arise in the course of embedding. In the version given here both positive and negative offsets are used, but the absolute value of the offset could also have been used, in which case the array size can be halved. Overcounting due to symmetry is a separate problem treated later.

Initialization (the first vertex is at a fixed origin) –

```
site_flag(*) ← empty
graph_count ← 0
vertex# ← 1
vertex_loc(1) ← origin
site_flag(origin) ← full
```

Add vertex –

```
/1/ ++ vertex#
index(vertex#) ← 0
```

Attempt to place latest vertex on a vacant lattice site –

```
/2/ edge# ← edge_total(vertex# - 1) + 1
repeat
    if index(vertex#) = lattice_degree then goto /3/
    ++ index(vertex#)
    vertex_loc(vertex#) ← vertex_loc(start_vertex(edge#))
    + lattice_vector(index(vertex#))
until site_flag(vertex_loc(vertex#)) = empty
```

Check other edges to this vertex from vertices already in place (these appear as consecutive entries in the edge list) –

```
while edge# < edge_total(vertex#) do
    ++ edge#
    if site_access(vertex_loc(vertex#))
```

...
- vertex_loc(start_vertex(edge#))) = 0 then goto /2/
endwhile

Flag lattice site if graph still unfinished –
if vertex# < graph_vertices then
   site_flag(vertex_loc(vertex#)) ← full
   goto /1/

Otherwise count graph embedding –
else
   ++ graph_count
   goto /2/
endif

Remove latest vertex and clear site flag for previous vertex (algorithm ends when only
initial vertex remains) –
/3/ − − vertex#
   site_flag(vertex_loc(vertex#)) ← empty
   if vertex# > 1 then goto /2/

This algorithm should be compared with the simple algorithm for SAW counting; the similarity
is apparent, the differences reflecting the more general connectivity of the graph. Variables
introduced here that require explanation are as follows:

vertex_loc – The lattice sites occupied by the vertices of the graph.
edge_total – Total number of edges in the graph that terminate on vertices up to and including
the specified vertex
site_access – Array in which a non-zero element indicates that the associated offset between
lattice sites corresponds to a lattice vector (introduced above).

The vectors "edge_total" and "site_access" are readily generated. Since closely related quanti-
ties are required by the improved counting algorithm (below) the generation technique can be
borrowed for use here.

As with previous algorithms we have found it expedient to sacrifice a limited amount of
efficiency (recall that this is still not be efficient version of the counting algorithm) to enhance
clarity. Certain steps in the computation are repeated unnecessarily, additional variables could
have been defined to reduce the references to arrays indirectly via other arrays, and so on. Note
also that the algorithm could be rewritten in recursive form; two kinds of recursion can be
introduced, one whose function is to add a vertex at each level of incarnation, the other to check
the feasibility of placing an edge to a previously positioned vertex.

4.3. The efficient approach

4.3.1. Outline

In the case of self-avoiding walks we saw how a certain amount of preparatory computation
was capable of producing a more efficient algorithm. The same is true for lattice constants but
the data that are required differ somewhat.
The process of embedding star graphs is split into two stages [12] by distinguishing between nodes (vertices with degree \( \geq 3 \)) and other vertices (degree = 2). An attempt is first made to place the nodes on lattice sites in such a manner that it is possible – though not certain – that they can be connected by bridges (i.e. chains of edges of specified length). The second step is the finding of all allowed combinations of bridge realizations for a particular arrangement of nodes. The bridge realizations are taken from a list of short self-avoiding walk segments, suitably arranged, that is produced prior to beginning the counting. The graph itself is usually given in the form of an edge list; as a preliminary step such a graph description must be converted into a form that fits the algorithm. The separate components of the overall algorithm are described in the following subsections.

4.3.2. Data organization

The organization of self-avoiding segments used for graph counting is not the same as that used in the SAW algorithm. There it was sufficient to classify the segments by length; for the present problem it is also necessary to group them according to the location of the last site of each segment relative to its starting point. This grouping is to allow efficient selection of a set of potentially useful segments for joining a pair of nodes already in position on the lattice. Thus an alternative procedure for producing segment tables is required, as well as additional pointers to permit access to the table based on new criteria not encountered in the SAW computation.

The segment table consists, as previously, of walk steps, and link variables associated with the walk steps which show where the next segment is to be found that experiences a change in the step concerned. Since the segments in this application are grouped according to endpoint, as well as by length, there is no need to include the final step of each segment because it is implicit in the means used for accessing the particular set of segments. Without loss of generality segments are assumed to begin at the origin; access to segments of given length terminating at a given lattice point is gained via a two-dimensional array that, for each endpoint position and segment length, gives the location of the initial relevant segment in the table – if such segments actually exist.

While the array of segment addresses just introduced provides immediate access to known segments, because of the fact that many array entries do not have corresponding segments – the array is sparse – an additional data structure is included to assist in the initial placement of the nodes. This is just a list of possible segment endpoints, ordered by segment length, and accompanied by pointers to the initial endpoint for each segment length. The final endpoint for each length is followed by a zero entry.

The two-dimensional array of segment addresses is called "site_access" and is the extension of a quantity of similar name introduced for the simple counting algorithm; there it was used to indicate the ability of a single edge to join two vertices of the graph already in place on the lattice, here it not only indicates whether a segment of specified length (one instance of which is a single edge) exists but where to find one or more potential candidates already tabulated. Where no suitable segments exist the corresponding array element is set to zero. The ancillary list of segment endpoints is denoted by "lattice_displace"; pointers to the first endpoint for each segment length are kept in "length_offset". Finally, the same two vectors as previously, "segment_step" and "segment_link", contain the segment details, although the actual step data stored differs because of the altered generation process; the initial link variables are again reserved for use as
4.3.3. Segment generation for graph counting

The algorithm for segment generation to be given below differs from the one described in the section on SAWs. The reason, as mentioned earlier, is that the segments must be grouped according to endpoint as well as length. Segments of given length are produced by adding a new initial step to suitably selected segments having a length that is one step shorter. The initial step always begins from the origin and the pre-existing segment is placed so that it starts from the endpoint of the first step; the only test necessary for violation of the self-avoidance condition is that none of the newly placed segment steps return to the origin, a test that is carried out directly rather than, as previously, by using flags that show which sites are occupied.

```
Initialization –
    site_access(*, *) ← 0
    segment_pointer ← max_length

Treat unit length segments (these are not stored explicitly) –
    for first_step = 1 to lattice_degree do
        site_access(lattice_vector(first_step), 1) ← 1
    endfor

    segment_link(0) ← 0

Longer segments –
    for length# = 2 to max_length do
        Loop over potential endpoints of segment –
            for last_site = −vector_limit*length# to vector_limit*length# do
                if last_site = origin then skip
                previous_pointer(1) ← length#

            Try possible first steps of segment –
                for first_step = 1 to lattice_degree do
                    site_loc(l) ← origin + lattice_vector(first_step)

                    Test whether endpoint accessible from site reached by first step –
                        index(1) ← site_access(last_site − site_loc(l), length# − 1) − 1
                    if index(1) = −1 then skip
                        saved_step(1) ← 0
                        step# ← 0

                    Add next step –
                        /1/  ++ step#
                        index(step#) ← index(step# − 1) + 1
                        go to /3/

                    Select next segment from table if available –
                        /2/  index(step#) ← segment_link(index(step#))
                        if index(step#) = 0 then goto /4/
```

pointers to the earliest table entries for each length but because an alternative method of accessing the segment table is used here these pointers have no function, their presence merely serving to simplify the segment generation algorithm.
Test for intersection at origin –
\[3/\] if \( \text{step} \# \leq \text{length} \# - 1 \) then
\[
\text{site} \_ \text{loc}(\text{step} \#) \leftarrow \text{site} \_ \text{loc}(\text{step} \# - 1) + \text{segment} \_ \text{step}(\text{index}(\text{step} \#))
\]
if \( \text{site} \_ \text{loc}(\text{step} \#) = \text{origin} \) then goto /2/\n\[2/\] goto /1/
endif

Find earliest changed step in completed segment –
if \( \text{saved} \_ \text{step}(1) \neq 0 \) then
\[
\text{step} \# \leftarrow 1
\]
repeat
\[
\text{++ step} \#
\]
until \( \text{saved} \_ \text{step}(\text{step} \#) \neq \text{segment} \_ \text{step}(\text{index}(\text{step} \#)) \)
\[
\text{changed} \_ \text{step} \leftarrow \text{step} \#
\]
else
\[
\text{saved} \_ \text{step}(1) \leftarrow \text{lattice} \_ \text{vector}(\text{first} \_ \text{step})
\]
\[
\text{step} \# \leftarrow 2
\]
\[
\text{changed} \_ \text{step} \leftarrow 1
\]
endif

Update changed steps –
while \( \text{step} \# \leq \text{length} \# - 1 \) do
\[
\text{saved} \_ \text{step}(\text{step} \#) \leftarrow \text{segment} \_ \text{step}(\text{index}(\text{step} \#))
\]
\[\text{++ step} \#\]
endwhile

Set pending pointer (belonging to a previous segment entry) and segment address array element –
\[
\text{segment} \_ \text{link}(\text{previous} \_ \text{pointer}(\text{changed} \_ \text{step})) \leftarrow \text{segment} \_ \text{pointer} + 1
\]
if \( \text{site} \_ \text{access}(\text{last} \_ \text{site}, \text{length} \#) = 0 \)
then
\[
\text{site} \_ \text{access}(\text{last} \_ \text{site}, \text{length} \#) \leftarrow \text{segment} \_ \text{pointer} + 1
\]

Save completed segment in table –
\[
\text{step} \# \leftarrow \text{changed} \_ \text{step} - 1
\]
repeat
\[
\text{++ step} \#
\]
\[
\text{++ segment} \_ \text{pointer}
\]
\[
\text{segment} \_ \text{step}(\text{segment} \_ \text{pointer}) \leftarrow \text{saved} \_ \text{step}(\text{step} \#)
\]
\[
\text{segment} \_ \text{link}(\text{segment} \_ \text{pointer}) \leftarrow 0
\]
\[
\text{previous} \_ \text{pointer}(\text{step} \#) \leftarrow \text{segment} \_ \text{pointer}
\]
until \( \text{step} \# = \text{length} \# - 1 \)
if \( \text{length} \# > 2 \) then goto /2/

Remove step –
\[4/\] \[\text{-- step} \#\]
if \( \text{step} \# > 1 \) then goto /2/
endfor
endfor
endfor
Build list of segment endpoints –
last_entry ← 0
for length# = 1 to max_length do
    length_offset(length#) ← last_entry + 1
    for last_site = −vector_limit∗length# to vector_limit∗length# do
        if site_access(last_site, length#) > 0 then
            ++ last_entry
            lattice_displace(last_entry) ← last_site
        endif
    endfor
Flag to indicate final entry for this length –
    ++ last_entry
    lattice_displace(last_entry) ← 0
endfor

The order in which the segments are produced is one in which the first step in a set of segments
to a given endpoint varies the least rapidly, the second step more rapidly, and so on. Thus the
possibility of the same initial step(s) appearing in consecutive segments arises and (as with the
SAWs) the use of pointers eliminates the need to retain this redundant information.

Variables introduced in this algorithm are the following (the remainder are as in the SAW
segment generation process):

site_access – Location in segment table of first segment to given endpoint with specified
length: for unit length only the feasibility of access is indicated (usage differs
from the simple counting algorithm).

first_step – Counter for trial initial steps.

last_site – Trial endpoint of segment.

vector_limit – Largest absolute value of lattice vector displacement.

index – Pointer to step location within segment table.

saved_step – Steps explicitly or implicitly saved from previously tabulated segment to
same endpoint.

last_entry – Latest value in segment endpoint list.

length_offset – Location of first entry in endpoint list for given segment length.

lattice_displace – List of segment endpoints ordered by number of steps in segment.

4.3.4. Revising the graph description

The graph description that is presented to the counting algorithm – typically the numbers of
vertices and edges together with a list of the vertex pairs joined by edges (the first member of
each pair can be assumed to be smaller than the second) – must be reduced to a form suitable for
manipulation. With a couple of notable exceptions this amounts to converting the graph back
into its homeomorphically irreducible form (viz., the topology) while keeping a record of the
bridge lengths. The exceptions to this statement are the following: First, the bridges cannot be
longer than the maximum available segment length; longer bridges must be subdivided by
reinserting vertices – which we will also refer to as nodes in this discussion to avoid confusion.
Second, the underlying lattice symmetry can considerably reduce the amount of counting required to evaluate a lattice constant; the initial vertex of the graph and two (typically, but more could be used) of its immediate neighbors are therefore given special treatment, the first step of which is to regard all three as nodes ('special' nodes) with the first being joined to the other two by bridges of unit length. The bridges must be ordered with respect to the nodes, following the same rule for edge sequencing used in the simple counting algorithm given previously.

Initialize, and select special vertices –

\[
\begin{align*}
\text{special\_vertex}(\ast) & \leftarrow 0 \\
\text{degree}(\ast) & \leftarrow 0 \\
\text{special\_vertex}(1) & \leftarrow 1 \\
\text{special}\# & \leftarrow 1 \\
\text{for edge}\# & = 1 \text{ to graph\_edges do} \\
\text{length}(\text{edge}\#) & \leftarrow 1 \\
\text{start\_node}(\text{edge}\#) & \leftarrow \text{start\_vertex}(\text{edge}\#) \\
\text{end\_node}(\text{edge}\#) & \leftarrow \text{end\_vertex}(\text{edge}\#) \\
\text{++ degree}(\text{start\_node}(\text{edge}\#)) \\
\text{++ degree}(\text{end\_node}(\text{edge}\#)) \\
\text{if special}\# < \text{total\_special} & \& \text{start\_node}(\text{edge}\#) = 1 \text{ then} \\
\text{++ special}\# \\
\text{special\_vertex}(\text{end\_node}(\text{edge}\#)) & \leftarrow 1 \\
\text{endif} \\
\text{endfor} \\
\text{Remove all non-special vertices of degree two –} \\
\text{for edge}\# = 1 \text{ to graph\_edges do} \\
\text{if length}(\text{edge}\#) = 0 \text{ then skip} \\
\text{if degree}(\text{start\_node}(\text{edge}\#)) = 2 \& \text{special\_vertex}(\text{start\_node}(\text{edge}\#)) = 0 \text{ then} \\
\text{erase\_vertex} & \leftarrow \text{start\_node}(\text{edge}\#) \\
\text{start\_node}(\text{edge}\#) & \leftarrow \text{end\_node}(\text{edge}\#) \\
\text{elseif degree}(\text{end\_node}(\text{edge}\#)) = 2 \& \text{special\_vertex}(\text{end\_node}(\text{edge}\#)) = 0 \text{ then} \\
\text{erase\_vertex} & \leftarrow \text{end\_node}(\text{edge}\#) \\
\text{else} \\
\text{skip} \\
\text{endif} \\
\text{Find the other edge incident on the vertex that is to be removed –} \\
\text{edge2}\# & \leftarrow \text{edge}\# \\
\text{repeat} \\
\text{++ edge2}\# \\
\text{until start\_node}(\text{edge2}\#) = \text{erase\_vertex} \text{ | end\_node}(\text{edge2}\#) = \text{erase\_vertex} \\
\text{if start\_node}(\text{edge}\#) = \text{erase\_vertex} \text{ then} \\
\text{start\_node}(\text{edge2}\#) & \leftarrow \text{start\_node}(\text{edge}\#) \\
\text{else} \\
\text{end\_node}(\text{edge2}\#) & \leftarrow \text{start\_node}(\text{edge}\#) \\
\text{endif} \end{align*}
\]
length(edge2 #) ← length(edge2 #) + length(edge #)
length(edge #) ← 0
degree(erase_vertex) ← 0
endfor

Compress vertex labels and edge list to recover topology description

graph_nodes ← 0
for vertex# = 1 to graph_vertices & degree(vertex#) > 0 do
  ++ graph_nodes
  node_label(vertex#) ← graph_nodes
endfor

graph_bridges ← 0
for edge# = 1 to graph_edges & length(edge#) > 0 do
  ++ graph_bridges
  length(graph_bridges) ← length(edge#)
  start_node(graph_bridges) ← node_label(start_node(edge#))
  end_node(graph_bridges) ← node_label(end_node(edge#))
endfor

Subdivide bridges that are too long –
for bridge# = 1 to graph_bridges & length(bridge#) > max_length do
  repeat
    ++ graph_bridges
    ++ graph_nodes
    start_node(graph_bridges) ← start_node(bridge#)
    end_node(graph_bridges) ← graph_nodes
    start_node(bridge#) ← graph_nodes
    length(graph_bridges) ← max_length
    length(bridge#) ← length(bridge#) − max_length
    until length(bridge#) ≤ max_length
endfor

Construct lists of node connections to facilitate relabeling –
degree( * ) ← 0
new_label( * ) ← 0
for bridge# = 1 to graph_bridges do
  ++ degree(start_node(bridge#))
  ++ degree(end_node(bridge#))
  connect(start_node(bridge#), degree(start_node(bridge#))) ←
  connect(end_node(bridge#), degree(end_node(bridge#))) ← ...
  start_node(bridge#)
endfor

Relabel nodes in order of appearance when scanning the lists of connections –
label# ← 1
new_label(1) ← 1
previous_label(1) ← 1
for node# = 1 to graph_nodes do
  for neighbor# = 1 to degree(previous_label(node#)) do
    neighbor_node ← connect(previous_label(node#), neighbor#)
    if new_label(neighbor_node) = 0 then
      ++ label#
      new_label(neighbor_node) ← label#
      previous_label(label#) ← neighbor_node
    endif
  endfor
endfor

Relabel bridges –
  start_node2(*) ← min(new_label(start_node(*)), new_label(end_node(*)))
  end_node2(*) ← max(new_label(start_node(*)), new_label(end_node(*)))

Produce additional data for counting algorithm and reorder bridges so that bridges terminating at a given node are listed in increasing length order –
  length2(*) ← length(*)
  length2(0) ← max_length + 1
  bridge# ← 0
  bridge_total(1) ← 0
  vertex_total(0) ← 0
for node# = 2 to graph_nodes do
  repeat
    Find shortest remaining bridge terminating at node –
      short_bridge ← 0
      for bridge2# = 1 to graph_bridges do
        if end_node2(bridge2#) = node# & length2(bridge2#) < length2(short_bridge)
          then short_bridge ← bridge2#
        endif
      endfor
      if short_bridge > 0 then
        ++ bridge#
        length(bridge#) ← length2(short_bridge)
        vertex_total(bridge#) ← vertex_total(bridge# - 1) + length(bridge#)
        length2(0) ← max_length + 1
        start_node(bridge#) ← start_node2(short_bridge)
      endif
    until short_bridge = 0
  bridge_total(node#) ← bridge#
endfor

If the topology description, rather than the graph description, had been supplied, the initial reduction to the topology could have been omitted, although the special vertices (if any) would still require attention. Note that the bridges are ordered according to their terminal nodes – this is the equivalent of the ordering requirement encountered in the simple graph counting algorithm.
expressed in alternate form. When there is more than one bridge having a common terminal node the bridges concerned are ordered by increasing length; a study of the logic of the counting algorithm will reveal why this is optimal for a given sequence of node labels.

A list of new variables used here follows:

- **special_vertex** – Indicates whether the vertex should be retained along with the nodes even though it may only be of degree two.
- **special** – Running count of special vertices.
- **total_special** – Number of special vertices.
- **erase_vertex** – Vertex to be removed because it is not a node.
- **graph_nodes** – Counts the nodes as they appear.
- **node_label** – Identifies graph vertex with node of topology.
- **graph_bridges** – Counts the bridges as they appear.
- **new_label** – New node label after relabeling.
- **connect** – A set of lists, one per node, of all nodes to which bridges exist.
- **label** – Running count of assigned node labels.
- **previous_label** – The node to which a new label corresponds.
- **neighbor_node** – A node taken from the list of neighbors “connect”.
- **start_node2** – Temporary storage for rearranged bridges.
- **end_node2** – As above.
- **length2** – As above.
- **bridge_total** – The total number of bridges that terminate on nodes up to and including the one referenced.
- **vertex_total** – The total number of vertices in bridges up to and including the one referenced; the initial node of each bridge is included in this count, but not the final node.
- **short_bridge** – The identity of the shortest bridge to a particular node.

A point worth commenting on is that the algorithm just described for converting the graph description to the required form is of comparable length to the counting algorithm itself (to follow). This is the price paid to attain efficiency. After studying the counting algorithm given in the next subsection it will also become all too apparent that the length of an algorithm (or the equivalent program need have little relationship with what the algorithm actually accomplishes. The relabeling could be done manually if necessary, even for complicated graphs; counting could not.

### 4.3.5. Fast graph counting – the initial phase

The initial phase deals with the placement of the special nodes on the lattice. For concreteness we assume that there are just three such nodes. The first is situated at the origin, the second at the end of one arbitrarily selected lattice vector (they are all equivalent), the third at the end of another (starting from the origin) which can be selected in several ways that are not equivalent under a symmetry transformation of the lattice. In the case of the fcc lattice for example, only four of the eleven possible positions of the third node are nonequivalent.
Placement of special nodes is the task of the outer loop of the counting algorithm, which is as follows. For convenience, the lattice vectors are assumed to be ordered so that the nonequivalent vectors (with respect to the first lattice vector) appear in sequence immediately following the first.

Initialization –

\[
\begin{align*}
site\_flag(\ast) & \leftarrow empty \\
total\_special & \leftarrow 3 \\
graph\_count & \leftarrow 0 \\
node\_loc(1) & \leftarrow origin \\
site\_flag(origin) & \leftarrow full \\
node\_loc(2) & \leftarrow origin + lattice\_vector(1) \\
\text{for node3 \# = 2 to possible\_vectors do} & \\
\quad node\_loc(3) & \leftarrow origin + lattice\_vector(node3 \#) \\
\quad place\_nodes & \\
\quad graph\_count & \leftarrow graph\_count + count \times node3\_factor(node3 \#) \\
\end{align*}
\]

endfor

Variables:

node\_loc – The lattice sites occupied by the graph nodes.
possible\_vectors – Indicates the last of the nonequivalent lattice vectors.
count – Number of embeddings returned by “place\_nodes”.
node3\_factor – Multiplier that accounts for the existence of equivalent lattice vectors.

The procedure “place\_nodes”, detailed below, computes the number of embeddings subject to the positions of the special nodes being fixed in advance. It will also handle the setting of site occupancy flags for special nodes beyond the first (and also clear the flag for the last of the special nodes before returning).

4.3.6. Node placement

This phase of the counting algorithm determines a feasible placement of all the nodes of the graph on suitable lattice sites. Feasibility implies that segments exist which can be used as bridges between the nodes, however there is no guarantee that there will not be any interference between bridges, or between bridge sites and nodes; this can only be determined during the third and final phase when the actual bridges are filled in. Before attempting to place additional nodes the algorithm checks to ensure that the special nodes already in place are positioned in a manner consistent with the segments available.

\[
\begin{align*}
\text{procedure place\_nodes:} \\
\text{Initialization –} & \\
\quad count & \leftarrow 0 \\
\quad node\# & \leftarrow 1 \\
\quad bridge\# & \leftarrow 0
\end{align*}
\]
Add node –
/1/ ++ node#
   if node# ≤ total_special then goto /3/
   displace_index(node#) ← length_offset(length(bridge# + 1)) – 1
This is where the procedure terminates once all non-special nodes have been removed in the course of backtracking –
/2/ if node# = total_special then return
Attempt to place latest node on vacant lattice site –
   bridge# ← bridge_total(node# – 1)
   repeat
      ++ displace_index(node#)
      if lattice_displace(displace_index(node#)) = 0 then goto /4/
      node_loc(node#) ← node_loc(start_node(bridge# + 1))
      + lattice_displace(displace_index(node#))
   until site_flag(node_loc(node#)) = empty
Check other bridges to this node from nodes already in place and save pointers to relevant blocks of entries in segment table –
/3/ repeat
      ++ bridge#
      segment_block(bridge#) ← site_access(node_loc(node#))
      – node_loc(start_node(bridge#)), length(bridge#) – 1
      if segment_block(bridge#) = –1 then goto /2/
      bridge_origin(bridge#) ← node_loc(start_node(bridge#))
   until bridge# = bridge_total(node#)
Flag lattice site occupied by latest node and test whether all nodes are in place –
   site_flag(node_loc(node#)) ← full
   if node# < graph_nodes then goto /1/
Place bridge vertice (if any) on lattice –
   place_bridges
Clear site occupied by final node –
   site_flag(node_loc(graph_nodes)) ← empty
   goto /2/
Remove node and clear site occupied by previous node –
/4/ –– node#
   site_flag(node_loc(node#)) ← empty
   goto /2/
endproc

Variables:

- displace_index – Pointer to list of segment endpoints.
- segment_block – Records the beginning location of a set of segments that will be used later when attempting to fill in bridge.
- bridge_origin – The starting site of the bridge.
The counting algorithm given here is limited to lattices in which all sites are equivalent – in other words a single set of lattice vectors describes the neighbor site positions relative to each site. There are lattices where this situation does not hold, and then two (and sometimes more) classes of lattice sites must be introduced, each with its own set of lattice vectors. We will not explore this aspect here; suffice it to say that a series of changes at various points in the algorithm are needed to distinguish between site classes. As mentioned earlier, for the lattices covered by the present version of the algorithm (those with only a single class of site) the size of the array “site_access” can be halved if it is indexed using the absolute value of the displacement between the starting and ending nodes of the bridge; the value inserted in “bridge_origin” must then be the lesser of the two node locations.

4.3.7. Filling the bridges

Once a suitable arrangement of nodes on the lattice has been found the task that remain is to fill in the bridges by systematically selecting all valid combinations of segments that can be used to connect the nodes in such a manner that they neither overlap one another nor the nodes already in place. Bridges of unit length do not require consideration since the earlier phase of the algorithm provided all the processing necessary.

```plaintext
procedure place_bridges:
  bridge# ← 0
  Add bridge –
  /1/  ++ bridge#
  vertex# ← vertex_total(bridge# - 1) + 1
  vertex_loc(vertex#) ← bridge_origin(bridge#)
  index(vertex#) ← segment_block(bridge#)
  if length(bridge#) > 1 then
    ++ vertex#
    index(vertex#) ← index(vertex# - 1) + 1
  endif
  goto /3/
  Select next segment if available –
  /2/  index(vertex#) ← segment_link(index(vertex#))
  if index(vertex#) = 0 then goto /4/
  Attempt to position segment vertices –
  /3/  if length(bridge#) > 1 then
    vertex_loc(vertex#) ← vertex_loc(vertex# - 1) + segment_step(index(vertex#))
    if site_flag(vertex_loc(vertex#)) ≠ empty then goto /2/
    if vertex# < vertex_total(bridge#) then
      ++ vertex#
      index(vertex#) ← index(vertex# - 1) + 1
      goto /3/
    endif
  endif
  endif
```
Test whether graph is still incomplete –
    if bridge\# < graph\_bridges then
      Flag sites occupied by vertices of bridge –
        for vertex\# = vertex\_total(bridge\# - 1) + 2 to vertex\_total(bridge\#) do
          site\_flag(vertex\_loc(vertex\#)) ← full
        endfor
      goto /l/
    endif
  Graph complete
    ++ count
    goto /2/
Remove latest vertex from bridge –
  /4/ -- vertex\#
    if vertex\# > vertex\_total(bridge\# - 1) + 1 then goto /2/
Remove empty bridge and test if all bridges are gone (procedure end) –
  -- bridge\#
    if bridge\# > 0 then
      Clear sites occupied by previous bridge –
        for vertex\# = vertex\_total(bridge\# - 1) + 2 to vertex\_total(bridge\#) do
          site\_flag(vertex\_loc(vertex\#)) ← empty
        endfor
      vertex\# ← vertex\_total(bridge\#)
      goto /2/
    endif
endproc

Additional variables used:

vertex\_loc -- Lattice sites occupied by graph vertices (as distinct from nodes).
index -- Pointer to step of segment stored in table.

For the majority of graphs (with the possible exception of those with all short bridges) it is this
final phase of the algorithm that performs most of the work. Its relative brevity is a consequence
of the elaborate preparatory work described earlier. A major gain in efficiency is due to the fact
that sites occupied by the final bridge do no need to be flagged, for the same reason as in the
SAW counting algorithm. This also explains why bridges are ordered so that, where possible,
their lengths increase: the more of the computation time spent dealing with just the final bridge
(rather than switching from one bridge to another) the better the performance of the algorithm.
An additional source of improved efficiency compared with the straightforward approach of
considering a single edge at a time is the fact that the nodes are placed in a manner that usually
allows them to be joined subsequently, thereby avoiding many avenues of exploration that turn
out to lead nowhere.
One possible improvement that was omitted from the present version of the algorithm in the cause of simplicity stems from the observation that whenever a bridge (excluding the last) is completed, all its sites are flagged. Similarly, when a bridge (not the last) is removed, all the sites it occupied are cleared. Obviously some sites that are cleared may need to be reflagged when the next segment recorded in the table is used for the bridge; by keeping track of which sites change between successive segments, with or without additional information that could be included along with the segment table, some of the extraneous work could be eliminated. However, since most of the effort is usually devoted to the final bridge, whose sites are never flagged, the modest gain in speed may not warrant this additional complexity. Another related modification (used in one implementation of the algorithm) which does affect the amount of work associated with the final bridge, is the way the pointers “segment_link” are created and interpreted. Instead of a zero entry being used to denote the need to backtrack to an earlier step, with the counting algorithm being left to discover the identify of this earlier step, the pointer could indicate the correct segment directly (zero would only be used to indicate that no more segments are available). It would then be necessary to include with each segment the explicit value of the number of steps recorded for the segment. Whether the additional data accesses required to utilize this quantity still permit a faster computation than the vertex-by-vertex backtracking used here is a matter which could be investigated. In any event, for the SAW problem treated earlier, such an approach does not appear productive because successive segments will generally differ from one another by just the final step, a consequence of the absence of ordering according to endpoint.

Other directions for improving efficiency without changing the basic structure of the algorithm are of a more general nature. As was mentioned in the discussion of SAW counting, the data and pointers in the segment table can be stored in the same vector as alternate quantities. One can go even further in order to reduce accesses to storage by packing each such pair of quantities into a single computer word and using masking operations to extract either the data or the pointer. Whether such an approach, which will most likely call for assembly language programming, results in a significantly faster program is very much a function of the design of the computer, and no general statement on the subject can be made.

4.4. Correcting for symmetry

Essentially all problems of interest deal with unlabeled graphs, whereas the way graphs are counted implies a certain ordering of the vertices and hence that the graphs are in fact labeled. The resulting count must therefore be divided by the symmetry number of the graph, which itself is just the order of the automorphism group. We will not include a description of the algorithm for obtaining the symmetry number since it is essentially the same as the procedure used earlier to partition edges and vertices to aid in generating topologies. The only change required is the replacing of the operation that marks equivalent vertices and edges every time an automorphism is obtained with an operation that increments the symmetry number by unity, starting from an initial value of zero. Upon termination of the procedure the resulting symmetry number is used in producing the final value of the lattice constant.
5. Graph weights for series expansion

5.1. Introduction

The series expansion of a quantity $\Psi$, typical examples including the free energy and susceptibility, for a system such as the Ising model, can be expressed in the form [24]

$$\Psi(\Lambda) = \sum_g (g; \Lambda) L(g).$$

(1)

In this summation $\Lambda$ denotes the particular lattice on which the spins are situated, the sum over $g$ ranges over all connected graphs – often subject to certain restrictions – that can be embedded in the lattice, $(g; \Lambda)$ is the lattice constant of the graph, and $L(g)$ is the weight, or contribution of $g$ to the expansion. The reason (1) is of any practical value is that $L(g)$ itself can be expanded as a power series in the usual high-temperature variable for the Ising model, $w = \tanh(1/T)$, where $T$ is the temperature in reduced units, and the leading order term of $L(g)$ involves a power of $w$ not less than the number of edges in $g$. By ordering the graphs according to their edge count it is possible to obtain the first $N$ coefficients of the expansion of $\Psi$ by terminating the sum over $g$ once those graphs with as many as $N$ edges have been included. We have already described the production of graphs of the type needed and the method used to evaluate their lattice constants; in this section we focus on the weight functions, taking the Ising susceptibility as a specific example.

The existence of a particular, rather subtle characteristic of $L(g)$ leads to an especially attractive means for its computation. The quantity $\Psi(G)$ can be evaluated for an arbitrary graph $G$, in other words for a set of Ising spins located at the vertices of $G$ interacting only along the edges of $G$ (note however that for a finite graph $\Psi(G)$ is the actual quantity whereas for an infinite lattice $\Psi(\Lambda)$ is taken to be the value per site). If $\Psi(G)$ is extensive in the sense that if $G$ and $G'$ are not connected $\Psi(G + G') = \Psi(G) + \Psi(G')$, then $L(G)$ can be computed as

$$L(G) = \Psi(G) - \sum_{g \subseteq G} (g; G) L(g),$$

(2)

where the sum is over all distinct (unlabeled) subgraphs of $G$ (excluding $G$ itself) and $(g; G)$ is the number of ways each can be embedded in $G$ (the computation of which is a minor extension of the algorithm for producing the graph symmetry number). While it is possible to use this formula repeatedly to generate $L(G)$ for successively larger graphs, the more effective alternative route is based on adding a label to the contribution of each edge of $G$ to $\Psi(G)$ and then performing a multivariable expansion of $\Psi(G)$ from whose terms those that contribute to $L(G)$ are easily identified. The effort required in manipulating the lists of graphs and counting the ways of embedding one graph in another is thereby replaced by a simple exercise in manipulating multinomials.

In the case of the zero-field Ising susceptibility $\chi$ we have

$$\Psi(G) \equiv \chi(G) = v(G) + 2 \sum_{g^{(2)} \subseteq G} w^{v(g^{(2)})}/1 + \sum_{g^{(0)} \subseteq G} w^{v(g^{(0)})},$$

(3)
for a graph $G$; here $v(g)$ denotes the number of vertices of $g$, while the sums are over all subgraphs (not just the connected ones) $g^{(0)}$ and $g^{(2)}$ containing, respectively, exactly zero and two vertices of odd degree. (For the record, the logarithm of the denominator is related to the free energy of $G$ and a similar treatment can be applied.) Since the smallest $g^{(0)}$ possible has at least three edges (viz., a triangle) a Taylor expansion of the denominator, retaining just a few terms, is sufficient for most purposes; thus

$$x(G) = v(G) + 2 \left( \sum_{g^{(2)}} w^{v(g^{(2)})} \right) \left[ 1 - \sum_{g^{(0)}} w^{v(g^{(0)})} + \left( \sum_{g^{(0)}} w^{v(g^{(0)})} \right)^2 - \ldots \right]. \tag{4}$$

Each subgraph $g$ contributes a term $w^{v(g)}$ to the unexpanded form of $x(G)$; the 'trick' is to replace $w^{v(g)}$ by $w_{i_1} w_{i_2} \ldots w_{i_{v(g)}}$ where $(i_1, i_2, \ldots, i_{v(G)})$ is the set of edges of the particular subgraph $g$. The multinomial expansion of $x(G)$ is now evaluated and all terms of the expansion that do not involve at least one appearance of every $w_i$ from $G$ are discarded. It is easily proved [24,25] that what remains, once the fictitious labels are removed from the $w_i$, is just the expansion of $L(G)$ obtainable by the more cumbersome subgraph subtraction process. The effective performance of such a calculation forms the subject of this section.

5.2. Data representation

In carrying out the multinomial expansion of $x(G)$ there are only two facts that must be recorded with each term generated; these are the order of the term (i.e., the power of $w$ after the labels are removed) and the identities of the individual $w_i$ that contribute at least once – multiple appearances count as one for this purpose. The order of the term can either be explicitly noted or can be implied by the manner in which the terms are grouped. Recording the appearances of individual edge contributions can be achieved by assign bits of a data word to correspond to the edges – the appearance of a set bit in a given position denotes the presence of the particular $w_i$ at least once in the term, while a zero bit indicates its absence. The manipulation of these bit patterns is carried out using modulo-2 arithmetic (i.e., the standard Boolean operations). As to the number of subgraphs for which storage must be set aside, there are a total of $2^c$ subgraphs with all vertices of even degree ($c$ being the cyclomatic number of $G$) and a similar number for each possible pair of odd vertices.

5.3. Weight computation

5.3.1. Subgraph generation

The first step of the computation is to generate two distinct lists of subgraphs of $G$ corresponding to the sums in the numerator and denominator of (3). Since each subgraph will be described as a bit pattern showing the edges that are present, with no edges appearing more than once at this stage, the two lists will in fact be completely equivalent to the sums appearing in (3). The subsequent stage in the weight computation will involve the polynomial manipulations implied by (4); these will be described in the next subsection.
Initialization

\[
\begin{align*}
\text{count\_even} & \leftarrow 0 \\
\text{count\_2odd} & \leftarrow 0 \\
\text{bit\_flag2}(\ast) & \leftarrow \text{bit\_flag(start\_edge(\ast))} \uparrow \text{bit\_flag(end\_edge(\ast))}
\end{align*}
\]

Loop over possible subgraph sizes

\[
\text{for subgraph\_edges} = 1 \text{ to graph\_edges do}
\]

Initial edge selection for subgraph

\[
\begin{align*}
\text{odd\_vertices} & \leftarrow 0 \\
\text{edges} & \leftarrow 0 \\
\text{for edge}\# = 1 \text{ to subgraph\_edges do}
\end{align*}
\]

\[
\begin{align*}
\text{odd\_vertices} & \leftarrow \text{odd\_vertices} \uparrow \text{bit\_flag2(\text{edge}\#)} \\
\text{edge\_loc(\text{edge}\#)} & \leftarrow \text{subgraph\_edges} + 1 - \text{edge}\# \\
\text{edges} & \leftarrow \text{edges} \uparrow \text{bit\_flag(edge\_loc(\text{edge}\#))}
\end{align*}
\]

Test whether all vertices are of even degree

\[
/1/ \quad \text{if odd\_vertices} = 0 \text{ then}
\]

\[
\begin{align*}
++ & \text{count\_even} \\
\text{subgraph\_even(count\_even)} & \leftarrow \text{edges} \\
\text{size\_even(count\_even)} & \leftarrow \text{subgraph\_edges}
\end{align*}
\]

Test whether there are exactly two odd vertices

\[
\text{elseif bit\_count(odd\_vertices) = 2 then}
\]

\[
\begin{align*}
++ & \text{count\_2odd} \\
\text{subgraph\_2odd(count\_2odd)} & \leftarrow \text{edges} \\
\text{size\_2odd(count\_2odd)} & \leftarrow \text{subgraph\_edges}
\end{align*}
\]

endif

Change edge selection to produce other subgraphs

\[
\text{edge}\# \leftarrow 0 \\
\text{repeat}
\]

\[
++ \text{edge}\#
\]

\[
\text{if edge\_loc(\text{edge}\#)} < \text{graph\_edges} + 1 - \text{edge}\# \text{ then}
\]

\[
\begin{align*}
\text{position}\# & \leftarrow \text{edge}\# \\
\text{repeat}
\end{align*}
\]

\[
++ \text{position}\#
\]

\[
\begin{align*}
\text{odd\_vertices} & \leftarrow \text{odd\_vertices} \uparrow \text{bit\_flag2(edge\_loc(\text{edge}\#))} \uparrow \text{bit\_flag2(position}\#)
\end{align*}
\]

\[
\begin{align*}
\text{edges} & \leftarrow \text{edges} \uparrow \text{bit\_flag(edge\_loc(\text{edge}\#))} \uparrow \text{bit\_flag(position}\#)
\end{align*}
\]

\[
\text{edge\_loc(\text{edge}\#)} \leftarrow \text{position}\#
\]

\[
- - \text{edge}\#
\]

\[
\text{until edge}\# = 0 \\
\text{goto} \ /1/
\]

endif

until edge\# = subgraph\_edges

endfor
Variables:

- `count_even` - Running count of subgraphs with all vertices of even degree.
- `count_2odd` - Running count of subgraphs with two odd vertices.
- `bit_flag` - Array of words with a single bit set is successively higher order positions; the `ith` element has value $2^{i-1}$.
- `bit_flag2` - Edge descriptors; two bits are set corresponding to the vertices on which the edge is incident.
- `subgraph_edges` - Current number of edges in subgraph.
- `odd_vertices` - Bits that correspond to odd degree vertices are set in this word.
- `edges` - Bits corresponding to currently selected edges are set here.
- `edge_loc` - List of currently selected edges.
- `subgraph_even` - List of subgraph descriptors (see “edges”) for even-vertex subgraphs.
- `size_even` - Number of edges in each even-vertex subgraph listed.
- `subgraph_2odd` - As for “subgraph_even”, but for subgraphs with two odd vertices.
- `size_2odd` - See above.
- `position` - Used to help change subgraph edges.

For counting the number of set bits in a word the procedure “bit_count” is used. This is sometimes an available system function and there may even be a hardware instruction to do the job; the alternative is to count bits, using predefined counts indexed by the bit patterns themselves to as great an extent as possible. For a reasonable number of bits (typically 12) a full set of counts can be prepared and stored, but for a greater number it will be necessary to divide the bit pattern into two or more parts, carry out a lookup for the individual partial counts and add the numbers.

In the algorithm the subgraph descriptors are paired with the actual edge counts. The latter could be eliminated if pointers to the start of each set of subgraphs with given edge count were maintained; this is a simple extension of the algorithm. It is assumed that the numbers of edges and vertices are not too large to prevent the use of a single word for each quantity; again, should this present a problem, suitable modification is not a major effort.

### 5.3.2. Expansion of the weight function

The second stage of the computation evaluates the terms of the weight expansion (4) and selects only those term which would not be eliminated in the course of subtracting the subgraph contributions as in (2). To convey the flavor of the algorithm we present a version that only generates terms as far as second order in the expansion of the denominator, as shown in (4); generalizing the computation to higher order is an easy exercise.

```
Initialization –
weight(*) ← 0
all_edges ← 2 ** graph_edges − 1
Consider all subgraphs contributing to numerator –
if subgraph_2odd(count_2odd) = all_edges then weight(graph_edges) ← 2
```
for n1# = 1 to count_2odd do
    edge_total1 ← size_2odd(n1#)
    edges_used1 ← subgraph_2odd(n1#)
Consider all subgraphs contributing to denominator for the linear term of the expansion –
for n2# = 1 to count_even do
    edge_total2 ← edge_total1 + size_even(n2#)
    if edge_total2 > max_order then break
    edges_used2 ← edges_used1 ++ subgraph_even(n2#)
    if edges_used2 = all_edges then weight(edge_total2) ← weight(edge_total2) - 2
Consider all subgraphs in denominator, yet again, for quadratic term of expansion –
for n3# = 1 to n2# do
    edge_total3 ← edge_total2 + size_even(n3#)
    if edge_total3 > max_order then break
    edges_used3 ← edges_used2 ++ subgraph_even(n3#)
    if edges_used3 = all_edges then
        value ← 4
        if n3# = n2# then value ← 2
        weight(edge_total3) ← weight(edge_total3) + value
    endif
endfor
endfor

New variables:

weight - The coefficients of the weight expansion for the graph.
all_edges - The bits for all edges in the graph are set here.
edge_total1/2/3 - Cumulative edge counts of terms generated in expansion.
edges_used1/2/3 - Cumulative records of all edges appearing at least once in the terms of
    the expansion.
max_order - The highest order term (i.e., power of w) of the expansion.

The ideas used here also find application in the expansion of weight functions for other series.
The free energy has already been mentioned. Another example is the second field derivative of
the susceptibility [25]. It should of course be pointed out that these techniques also apply to
generalizations of the Ising model such as the dilute systems (either spins or interactions missing
at random with prescribed probability) and to spin-glass models (e.g., with interactions between
neighbor spins of random sign, again with given probability) [25,26]. Another use of the weight
expansion is for the inverse susceptibility expansion [27,28]; this is a more complex calculation
which has the additional requirement of classifying the subgraphs with two odd vertices
according to the identities of the vertices themselves.
6. Self-avoiding walks – the Monte Carlo approach

6.1. The problem

The Monte Carlo method for generating self-avoiding walks produces a representative sample of walks from which the configurational properties can be measured directly. If the walks are of sufficient length the asymptotic behavior should (almost) be observable directly without resorting to the extrapolations needed subsequent to exact enumeration. The problem faced in trying to generate a walk at random which does not intersect itself is one of attrition – the vast majority of walks started terminate prematurely. Any solution to this fundamental difficulty must avoid introducing bias into the sample; it is forbidden, for example, to simply try another step direction whenever a blockage is encountered since this immediately leads to bias – the sample distribution has become distorted.

Attrition can be overcome by means of enrichment [29]. As originally formulated the enrichment approach attempts to add a fixed number of steps, \( s \), to the walk under construction. If it is successful it proceeds further, but if an intersection occurs all of the latest steps are discarded and a new attempt is made to add an \( s \)-step sequence. The preceding sequence of \( s \) steps is used exactly \( p \) times in attempts (whether successful or not) to extend the walk a further \( s \) steps, and then it too is discarded and an attempt made to find a replacement sequence. The two parameters \( p \) and \( s \) are related, and are chosen in such a manner as to achieve a reasonable walk completion rate, while at the same time endeavoring to prevent the situation where the same early steps of the walk are present in a disproportionate fraction of the final walk configurations.

The method described below enlarges upon this technique. Walks are generated by means of a series of three distinct processes. The first of these is exact enumeration, and the latter two are based on the enrichment technique, with each using the output of the preceding process for its input. Different sets of enrichment parameters are used in the two applications of enrichment because of the different walk lengths involved. A further change is not to insist on a single value of \( p \) but to use a set of \( p \) values that are applied in fixed order as the sequences of steps are added; when one starts to generate very long walks the success rate (and the reappearance rate of the early step sequences) are found to depend strongly on \( p \) (for a given \( s \)), and having several such values that can be adjusted makes it easier to finely tune the enrichment process. Results obtained recently using this approach are described in ref. [10].

6.2. Outline of method

Walks produced using the Monte Carlo method can be two orders of magnitude longer than the walks involved in exact enumeration; instead of walks with, typically, 10–20 steps, we are now dealing with walks that may exceed 2000 steps in length. Much of the efficiency of the exact enumeration algorithm described earlier was due to the use of a lattice to permit rapid testing for self-interaction: instead of having to compare the \( N \)th site visited by the walk with each of the preceding \((N-1)\) sites, a direct query addressed to the lattice site concerned produced an immediate answer to the occupancy question. For the comparatively short walks dealt with by
exact enumeration the lattice itself was of reasonable size and could be represented in the
computer without difficulty; on the other hand, to handle a walk of 2000 steps in the same way
would necessitate a lattice of 4000^3 sites which is obviously out of the question.

A more reasonable line of attack, one that retains the all-important lattice while avoiding its
excessive size, is to map the lattice – in a many-to-one manner – onto a much smaller domain. If,
for example, the domain is a cube with 2^5 sites on an edge (i.e., a total of 2^15 sites in three
dimensions), and the edges of the lattice itself each extend over 2^13 sites, then a total of 2^24 sites
of the original lattice map to each site of the reduced domain. Each of the longest walks that will
be generated visit of the order of 2^11 sites, considerably fewer than the number in the domain.
While the possibility exists that a walk will occupy several distinct lattice sites that map to the
same site in the domain, the probability that such an event will occur is small; awareness of this
fact helps in devising a method for keeping track of the sites visited by the walk.

The method chosen requires that the different domain-sized regions of the lattice be suitably
numbered and, as the walk is laid down on the lattice, that a record be maintained as to which
region serves as the current one. These region numbers are stored in a vector associated with the
steps of the walk, and distinct lattice sites that map to the same site of the domain are linked
together. Rapid access to these linked lists is provided by an array of pointers indexed by the
occupied sites of the domain. Only when two or more lattice sites visited correspond to a single
domain site will it be necessary to traverse the linked sequence of region numbers to determine
whether an intersection really occurs, otherwise, and this is the most likely scenario, access to the
region number is almost immediate. Further details accompany the description of the data
structures that are used by the algorithm.

The way the enrichment technique is employed here differs from the original application. The
walks are generated via a three-stage process. The first stage is the construction of a list of all
short SAWs of a given length – segments – just as was done for exact enumeration. The second
stage is to use enrichment Monte Carlo to combine segments into intermediate lengths SAWs,
which for convenience will be referred to as fragments; these are still not the final walks. (Note a
slight change of terminology from an earlier brief description of the computation [10] which used
the term ‘element’ in place of ‘segment’, and ‘segment’ instead of ‘fragment’; the change is to
maintain consistent use of the term ‘segment’.) The third and final stage again involves
enrichment, on this occasion to combine fragments to produce the desired length of walk.
Typical lengths of segments and fragments are six and sixty, while the full walks can be as much
as 2400 steps in length. The enrichment parameters corresponding to p are chosen by experiment-
tation to ensure both that walks actually emerge from the generation process and that the same
initial segments or fragments (depending on the stage) do not appear too frequently in successive
fragments or walks that are produced.

We will describe only the walk generation stage of the algorithm in detail to avoid too much
repetition. Generation of the segments is the same as for exact enumeration, although the data is
stored in simpler fashion (see further). The technique used to produce the fragments is based on
the one used for the walks themselves; it is a simplified version of the latter with a few of the
computational steps removed and a new step inserted that stores the fragments as they are
produced. There should be little difficulty producing such an algorithm given the version that
generates the final walks.
6.3. Data organization

In common with other algorithms described in this paper a carefully chosen structuring of the data required or produced by the calculation contributes both to the efficiency and (hopefully) the intelligibility of the algorithm. Several categories of data play key roles in the SAW Monte Carlo algorithm, including tables of segments and fragments, data to enable a walk in progress to be modified when it is exiting the current lattice region (corresponding to a particular mapping to the domain) as well as the identity of the new region about to be entered, and the linked data structure mentioned earlier that enables the detection of walk intersections, but nevertheless allows multiple lattice sites to be represented by a single domain site. These and allied matters are covered here.

The full set of segments of chosen length – no steps are omitted and there are no pointers as in the case of SAW enumeration – is stored in the segment descriptor table. One additional difference is that it is not the values of the lattice vectors that are kept but indices that specify the vectors; the reason for this change is to allow subsequent rearrangement of the lattice vectors (subject to symmetry considerations) that effectively enlarges the size of the set of fragments that are produced subsequently. The list of fragments is stored in the fragment descriptor table; since fragments are constructed from segments it is enough to store the sequence of identifiers of the segments used in building each fragment and thereby gain an order of magnitude reduction in storage requirements for a very small additional computational overhead.

Site occupancy is recorded in the usual way by means of an array in which one element is assigned to each site of the lattice domain. In this application however, the array of site flags performs an additional function beyond merely signifying whether a site is or is not occupied. The site flags corresponding to occupied sites now identify the most recent step of the walk to visit the site. If there are several such visitations (occurring in different domain-sized regions of the lattice) a linked list is constructed in a vector whose elements are associated with the walk steps, and this permits the examination of the region number in which the earlier visit(s) occurred.

In order to simplify the task of deciding whether the walk is about to exit a particular domain-sized region, a second array with size equal to the number of sites in the domain is set aside for the purpose of holding a set of boundary flags. Words in this array that correspond to sites at the boundary faces of the domain have an appropriate bit set; words for sites at edges (corners) of the domain have two (three) bits set corresponding to the faces involved. For a three-dimensional lattice six bits suffice; the boundary faces correspond to the $\pm x$, $\pm y$ and $\pm z$ extremes of the domain.

When the walk reaches a site at the domain boundary the next step taken may or may not lead to a change of region. A simple method of deciding the issue is to associate with each of the lattice vectors a word which contains bits set in the positions corresponding to the boundary face(s) that a step in the direction of the vector is capable of crossing; in the case of the simple cubic lattice for example, the vector in the $+y$ direction is only capable of causing a change of region if it crosses the $+y$ boundary. For lattices which have vectors that are not perpendicular to the boundary faces the situation is slightly more complicated, in that a vector may produce a simultaneous crossing of more than one boundary face.

When a boundary is crossed not only does the lattice region change but the next site of the
walk must be shifted to the opposite side of the domain (there is a formal similarity with periodic boundaries, but this is not to be taken literally). Each lattice vector must therefore be accompanied by two additional data items, namely the change in region number that occurs whenever the vector actually causes a transfer to a new region and the amount the walk must be displaced so that it appears to reenter the domain from the opposite face.

Specifying the above-itemized quantities is an uninspiring task. A brief summary of the process for the simplest three-dimensional lattice, the simple cubic, is as follows: The lattice is mapped onto the integers $1$ through $L^3$ so that the lattice vectors $+x$, $+y$, $+z$ correspond to increments of $1$, $L$, $L^2$ in the site numbers. The corners of the domain are at sites $1$, $L$, $L^2 - L + 1$, $L^2$, $L^3 - L^2 + 1$, $L^3 - L^2 + L$, $L^3 - L + 1$ and $L^3$; there should be little difficulty in setting the boundary flags given this information. Now assume that the regions of the lattice that map to the domain are arranged in the same way as the sites themselves, only this time as an $M^3$ array. The change in region number produced by crossing a $+x$, $+y$ or $+z$ face is then $1$, $M$ or $M^2$, respectively; the corresponding adjustments to the domain site number are $-L$, $-L^2$ or $-L^3$.

We mentioned earlier that the set of walk fragments could be enlarged by a suitable rearrangement of the lattice vectors. Each time a fragment is picked from the table a random selection is made among the six possible permutations of the $x$, $y$ and $z$ directions and among the eight possible combinations of reflections in the $x$-$y$, $y$-$z$ and $z$-$x$ planes. Given the fact that the fragments tabulated are only a very small proportion of the total possible, the rearrangement is tantamount to a 48-fold increase in the size of the fragment collection. Under these 48 transformations the lattice vectors exchange identities; a table giving these relationships is produced to permit the indices of the lattice vectors used in the fragments to be easily exchanged under the particular transformation chosen. It is for this reason that indices are stored in the segment table rather than actual displacements.

### 6.4. Walk generation from fragments

For reasons stated earlier this is the only part of the generation algorithm which will be presented in detail; fragment generation follows a similar procedure, while the segments are constructed by the now-familiar exact enumeration approach. The version given here is for lattices where only a single boundary face can be crossed per walk step (i.e., square, simple cubic, hypercubic).

Initialization –

```plaintext
site_flag(●) ← empty
walk_count ← 0
site_loc(0) ← origin
site_flag(origin) ← 0
site_link(0) ← empty
lattice_domain(0) ← origin_domain
fragment # ← 0
```
Prepare the add another fragment to walk –
/1/    ++ fragment#
    enrich_count(fragment# - 1) ← 0
Update enrichment count for preceding fragment and check that further usage is allowed –
/2/    if enrich_count(fragment# - 1) = enrich_limit(fragment# - 1) then goto /3/
    ++ enrich_count(fragment# - 1)
Randomly select next fragment from table and permute lattice vector indices –
fragment_end ← random_value(1, totalragments) * fragment_segment
fragment_orient ← random_value(1, 48)
segment# ← fragment_end – fragment_segments
site# ← (fragment# - 1) * fragment_steps
Add each segment of fragment –
repeat
    ++ segment#
    segment_end ← fragment_desc(segment#) * segment_steps
    step# ← segment_end – segment_steps
Add steps of segment assuming, tentatively, no domain change –
repeat
    ++ step#
    ++ site#
    vector_index ← orient(fragment_orient, segment_desc(step#))
    site_loc(site#) ← site_loc(site# - 1) + lattice_vector(vector_index)
    lattice_domain(site#) ← lattice_domain(site# - 1)
Test whether change of domain has actually occurred –
    if boundary_flags(site_loc(site# - 1)) * vector_flags(vector_index) ≠ 0 then
        lattice_domain(site#) ← lattice_domain(site#) +
        domain_change(vector_index)
        site_loc(site#) ← site_loc(site#) + site_change(vector_index)
    endif
Test whether this lattice site is already occupied –
    if site_flag(site_loc(site#)) ≠ empty then
        previous# ← site_flag(site_loc(site#))
        repeat
            if lattice_domain(previous#) = lattice_domain(site#) then goto /2/
            previous# ← site_link(previous#)
        until previous# = empty
    endif
Save lattice-vector index –
    step_record(site#) ← vector_index
    until step# = segment_end
    until segment# = fragment_end
Flag occupied sites (ensuring proper linkage) if not final fragment –
    if fragment# < walk_fragments then
        step_end ← fragment# * fragment_steps
step # ← step_end − fragment_steps
repeat
    ++ step#
    site_link(step#) ← site_flag(site_loc(step#))
    site_flag(site_loc(step#)) ← step#
until step# = step_end
goto /1/
endif
Compute properties of completed walk –
++ walk_out
compute_properties
Generation terminates when required number of walks have been produced –
if walk_count = walk_limit then exit
goto /2/
Remove latest fragment and clear sites occupied by previous fragment –
/3/ ← ← fragment#
step# ← fragment# * fragment_steps
step_start ← step# − fragment_steps
repeat
    site_flag(site_loc(step#)) ← site_link(site_flag(site_loc(step#)))
    −− step#
until step# = step_start
goto /2/

Key variable used in the algorithm not encountered previously are defined below in order of appearance:

site_flag – Indicates whether sites of lattice domain are occupied or empty; if occupied then the array element points to the walk step that reached the site last (the value "empty" must be distinct from the walk site numbers, typically −1).

site_link – Linkage variables that associate walk steps visiting different lattice sites that map to the same domain site; the last (or only) link of such a sequence has the value "empty".

lattice_domain – The domain number in which the walk step is located.

origin_domain – A starting value for the domain number (typically midrange).

enrich_count – Running count of the number of times a fragment is reused consecutively in attempts to extend the walk by adding a further fragment.

enrich_limit – Maximum allowed value of "enrich_count"; a fragment must be discarded when this limit is reached (the zeroth element should be set to −1 to ensure consistent operation of the algorithm).

fragment_end – Location of final entry for fragment in descriptor table.

total_fragments – Number of fragments in descriptor table.

fragment_segments – Number of segments per fragment.
The vector "step_index" contains a complete description of the generated walk. All properties of the walk can be deduced from this vector; here we will show how the squares of the end-to-end distance and radius of gyration are evaluated. These quantities were introduced previously in connection with SAW exact enumeration.

Recall the definitions (section 2) –

\[ R_N^2 = \langle (r_N - r_0)^2 \rangle, \]  

where the quantity whose configurational average is evaluated is

\[ (r_N - r_0)^2 = \sum_{xyz} (x_N - x_0)^2 \]
and

\[ S_N^2 = \frac{1}{N+1} \left( \frac{\sum_{i=0}^{N} r_i}{N+1} - \left[ \frac{1}{N+1} \sum_{j=0}^{N} r_j \right]^2 \right) \]  

(3)

where the quantity to be averaged is

\[ \sum_{xyz} \left( \frac{1}{N+1} \sum_i x_i^2 - \frac{1}{(N+1)^2} \left[ \sum_j x_j \right]^2 \right). \]  

(4)

In order to evaluate these quantities we first construct a matrix "displacement_matrix" whose columns are the cartesian components of the displacement produced by steps along each of the lattice vectors. The procedure is then as follows, assuming the walk to start at the origin.

procedure compute_properties:
Initialize –
  offset( *) ← 0
  total_displace( *) ← 0
  total_displace2 ← 0
Accumulate displacements needed for sums –
  for component # = 1 to 3 do
    for step # = 1 to walk_length do
      offset(component #) ← offset(component #) + ... offset(component #) +
      displacement_matrix(component #, lattice_vector(step_record(step #)))
      total_displace(component #) ← ...
      total_displace2(component #) ← total_displace2 + offset(component #) ** 2
  endfor
endfor
Produce final estimates for quantities – (use equations above)
endproc

Variables:

offset – Coordinates of sites visited (which respect to origin).
total_displace – Cumulative sums \( \sum x_i \), etc.
total_displace2 – Cumulative sum \( \sum x_i^2 \).
component # – Cartesian component.
displacement_matrix – Contains components of displacement corresponding to each vector (see text).
7. Percolation by Monte Carlo

7.1. Background

The percolation problem provides yet another example of a highly simplified model that finds
center in a variety of physical situations [32–34]. We shall restrict our discussion to the
problem of site percolation on two-dimensional lattices. Sites are occupied with probability \( p \)
and are vacant otherwise. Adjacent occupied sites are said to be connected, and in any random
assignment of occupied sites the formation of connected clusters will be observed. This leads to a
host of problems that can be studied; the one dealt with here is how cluster sizes are distributed.
As with previous problems the interest here is also dominated by asymptotics – what is the form
of the cluster-size distribution for large clusters when \( p \) is close to its critical value \( p_c \) at which
point the infinite cluster first puts in an appearance?

Large clusters can only reside on large lattices and it is therefore to be expected that long
computations are again in the offing. But, unlike the problems discussed so far, the accommoda-
tion of periodic boundaries is a far from trivial undertaking. This investment of effort is
however essential because the large-cluster end of the size distribution is seriously distorted by
clusters truncated at the lattice boundary if open boundaries are used; periodic boundaries
provide a closer approximation to the situation existing in the infinite lattice.

The sheer size of the lattice on which the study is carried out can lead to problems of
insufficient storage space in the computer as well as the impossibility of using integer quantities
to represent cluster size. The former problem has to be overcome somehow; the latter, though
not a fundamental obstacle, would lead to marked degradation in efficiency since the alternative
is to use floating-point arithmetic which is generally markedly slower than integer arithmetic.
Both difficulties are resolved when, instead of confronting the largest lattices head on, the
problem is divided into a series of smaller problems, each of which is dealt with separately, and
the final results combined. This is the approach to be described here; the results obtained are to
be found in ref. [35].

7.2. Techniques for cluster construction

7.2.1. The basic method

As occupied sites are detected during the process of scanning the lattice they either constitute
the initial sites of what are tentatively regarded as new clusters, or they are assigned to previously
existing clusters to which they are connected. In the event that an occupied site is connected to
two clusters which, up to that moment, had been treated as distinct, then the two must be
merged and the new site added to the combined cluster. In the basic approach to cluster
manipulation all sites belonging to (say) the smaller of the two clusters are reassigned to the
larger. As will be shown in the next subsection, much of the effort expended in reassigning sites
to alternative clusters is totally unnecessary and thus the simple approach is not suited to large
percolation studies. Nevertheless we will include the basic algorithm to permit the reader, for one
last time, to contrast a straightforward but inefficient approach with one whose capability is well
established.
The version of the basic algorithm given here assumed the existence of site pairs connected by bonds; each cluster is then a network of sites all accessible from one another by paths through the bond network. The algorithm is readily adapted (although as already indicated, not ideally suited) to the percolation problem; the version shown here was used in a study of hydrogen bond networks in water [36] - not a lattice problem but this fact does not affect the algorithm.

Initialize –
  cluster_id(*) ← 0
  total_clusters ← 0

Consider each bond in turn –
  for bond # = 1 to total_bonds do
    site1 ← start_site(bond #)
    site2 ← end_site(bond #)
    if cluster_id(sitel) = 0 then
      if cluster_id(site2) = 0 then
        Begin new cluster –
        ++ total_clusters
        cluster_id(sitel) ← total_clusters
        cluster_id(site2) ← total_clusters
        cluster_head(total_clusters) ← site1
        cluster_link(sitel) ← site2
        cluster_link(site2) ← null
        cluster_size(total_clusters) ← 2
      else
        Add “sitel” to cluster –
        cluster_id(sitel) ← cluster_id(site2)
        cluster_link(sitel) ← cluster_head(cluster_id(site2))
        cluster_head(cluster_id(site2)) ← site1
        ++ cluster_size(cluster_id(site2))
      endif
    else
      if cluster_id(site2) = 0 then
        Add “site2” to cluster –
        cluster_id(site2) ← cluster_id(sitel)
        cluster_link(site2) ← cluster_head(cluster_id(sitel))
        cluster_head(cluster_id(sitel)) ← site2
        ++ cluster_size(cluster_id(sitel))
      elseif cluster_id(site) ≠ cluster_id(site2) then
        Merge smaller cluster into larger –
        larger ← cluster_id(sitel)
        smaller ← cluster_id(site2)
        if cluster_size(cluster_id(sitel)) ≤ cluster_size(cluster_id(site2)) then
          larger ↔ smaller

      ...
Update cluster membership –
  site# ← cluster_head(smaller)
repeat
  cluster_id(site#) ← larger
  final_site ← site#
  site# ← cluster_link(site#)
until site# = null
Relink to complete merge –
  cluster_link(final_site) ← cluster_head(larger)
  cluster_head(larger) ← cluster_head(smaller)
  cluster_size(larger) ← cluster_size(larger) + cluster_size(smaller)
  cluster_size(smaller) ← 0
endif
endif
endfor
Construct histogram of cluster sizes (unit bin width) –
  bin(*) ← 0
  bin(1) ← total_sites
  for cluster# = 1 to total_clusters & cluster_size(cluster#) > 0 do
    ++ bin(cluster_size(cluster#))
    bin(1) ← bin(1) − cluster_size(cluster#)
  endfor

Variables needing explanation are as follows:

cluster_id – Shows the cluster to which the site belongs.
total_clusters – Number of new clusters; some are merged subsequently.
cluster_head – Pointers to first site in each cluster.
cluster_link – Pointers to successive sites in clusters; last site has “null” value for pointer (e.g., zero).
cluster_size – Number of sites in cluster.
bin – The resulting cluster size histogram; note that the first element is the number of isolated sites.

While the above algorithm is simple to use and effective for small problems, the work required for cluster merging in larger problem becomes prohibitive. The alternative scheme described below avoids this extra work.

7.2.2. Multiple labeling

The way to eliminate the laborious process of reassigning sites each time a cluster merger takes place is by allowing each cluster to have many labels, one of which is chosen as the true label, the others as aliases. Supplementary data accompanying each alias allows retrieval of the true label of a cluster whenever this information is required.
The method can be summarized as follows \cite{37}: From a pool of available labels assign an unused label to each new cluster as it appears. If, at some subsequent stage, two such clusters are merged, then the label of one of them is converted into an alias for the other. In practice there exists an array of quantities \( \{ E_m \} \) that are used to contain either the sizes of clusters or links between different labels belonging to the same cluster. If \( m \) happens to be a true label then the quantity \( E_m \) denotes the size of the cluster at a given stage of the generation process; if, on the other hand, \( m \) is only an alias, then \(-E_m\) is interpreted as a pointer to either the true label or to yet another alias for the same cluster. As the generation proceeds subsets of the \( \{ E_m \} \) will form pointer chains to be true labels of the clusters present on the lattice. In theory these chains could become quite long, but in practice, close to the threshold \( p_c \) (where the interesting physics is to be found) the chains are extremely short, implying that minimal effort is spent searching for true labels. The pool of labels must be replenished from time to time (the frequency depending on the pool size); this is done by releasing all aliases, as well as those labels assigned to clusters whose growth has terminated and which can therefore be added to the summary of whatever cluster properties are being evaluated. A program that implements the method as described so far, with open boundaries, appears in ref. \cite{38}.

7.2.3. Slab subdivision

The multiple labeling method stores both cluster sizes and label pointers in the same array. While this is not essential it obviously economizes on space. It does however restrict the cluster sizes in that they cannot exceed the largest integer the particular computer can handle. The alternative would be to store cluster sizes as floating-point quantities of suitable precision, but this would not only consume extra space but also slow down the computation. It is obviously desirable to continue to represent cluster sizes with integer variables while at the same time allowing the huge cluster sizes envisaged for the present computation to exceed the capacity of the normally encountered 32-bit integer (whose maximum value is limited to \( 2^{31} - 1 \)).

The way to accomplish this goal is to subdivide the lattice into regions – called slabs – that are small enough to ensure that the largest cluster that can be grown is of size less than the limit set by the computer word length. The clusters present on the full lattice can be recovered by joining these slabs together in a calculation whose duration is insignificant compared with that required to generate the slabs and for which there is therefore no objection to spending more time and space on dealing with clusters that exceed the integer limit. Two additional advantages are that the individual slab computations, though time consuming, do not require a great deal of storage – computer charges often include a consideration for storage as well; the other is that the slabs can be combined in different ways, thereby achieving at no cost the same effect as the considerably more expensive production of several distinct copies of the entire lattice. The price of these gains is, as ever, one of increased algorithmic complexity.

As pointed out earlier, the lattices being dealt with are indeed very large – the biggest a square region containing \( 2.56 \times 10^{10} \) sites. The slabs are smaller and contain a mere \( 10^8 \) sites each, small enough to keep the cluster sizes well within the allowed range. The largest full lattice thus requires the joining of 256 slabs, and it is not hard to foresee that a substantial amount of bookkeeping will be required to support this activity. We will return to this matter later.
7.3. Slab generation algorithm

7.3.1. Overall structure

A lattice is generated by scanning the sites a row at a time, deciding whether or not each site is occupied and, if the answer is in the affirmative, carrying out whatever is needed to ensure that the clusters affected grow correctly. An interesting feature of the row-by-row approach is that only the sites of a single row need be stored; the row generally consists of two parts – the sites of the partially generated latest row and sites of the previous row that are adjacent to those of the latest row that have yet to be examined. In fact the design of the algorithm allows the latest row to overwrite its predecessor in storage. Since any information that could be learned from previous rows is saved along with the clusters to which it relates, there is no value in retaining details of previous rows and thus only a single row, the most recent, is kept. The exception to this statement is the set of sites on the slab boundary; in order to support the subsequent joining of different slabs and/or the application of periodic boundary conditions the peripheral sites must be retained. If the slab had open boundaries and no joining was intended then the sites of the latest row are all that is required.

Sites are filled with probability \( p \); again, to avoid the need for floating-point arithmetic, the value of \( p \) is scaled by multiplying it with a suitably large number (e.g., \( 2^{24} \)) and converting the result to the nearest integer. The only cluster properties extracted are the sizes; with a little more effort the same algorithm could yield other properties, although not those whose measurement requires all the sites of the cluster to be present simultaneously. Finally, the algorithm is tailored to the square lattice; the minor modification needed to adapt it for the triangular lattice amounts to considering two neighbor sites in the previous row rather than just one.

Initialization –

```
bin(*) ← 0
row_site(*) ← 0
cluster_count ← 0
```

Consider each row of slab –

```
for row# = 1 to slab_edge do
    generate_row
    Save perimeter sites of current row –
        perimeter_site(row#, 2) ← row_site(slab_edge)
        perimeter_site(row#, 4) ← row_site(1)
    Save entire row if first or last –
        if row# = 1 | row# = slab_edge then
            boundary# ← 1
            if row# = slab_edge then boundary# ← 3
            perimeter_site(*, boundary#) ← row_site(*)
        endif
    Compress labels if necessary (the compression procedure acts on the array “perimeter_site”) –
        if cluster_count > nearly_full then
            perimeter_site(*, 3) ← row_site(*)
```


compress_labels
row_site(*) ← perimeter_site(*, 3)
endif
endfor
Final compression –
compress_labels
Save results –
store_slab

Variables introduced so far:

bin - Cluster-size histogram counts
row_site - Sites of latest row – either indicates that site is unoccupied (= 0) or gives information about cluster label; indexing begins at zero.
cluster_count - Number of cluster labels currently in use.
row# - Current row of lattice.
slab_edge - Number of sites per slab edge.
perimeter_site - Copy of site information ("row_site") for perimeter sites.
nearly_full - When "cluster_count" exceeds this threshold compression becomes due.

7.3.2. Row processing

The processing of each row of the slab is carried out by the following procedure. Merging of clusters will be seen to be a very minor operation because of the use of multiple labels (true labels and aliases). Note that while the random number generating procedure is called separately for each site, in the operational version of the program a single invocation produces enough random numbers for the entire row (this is the efficient way, especially for shift-register generators).

procedure generate_row:
Scan all sites of current row –
for site# = 1 to slab_edge do
Test for empty site –
if random_value(0, scale_factor) > occupation_probability then
row_site(site#) ← 0
Occupied site –
elseif row_site(site#) = 0 & row_site(site# - 1) = 0 then
Site with no neighbors so far, so create new cluster –
++ cluster_count
row_site(site#) ← cluster_count
cluster_desc(cluster_count) ← 1
else
Occupied site with neighbor(s), determine true label(s) –
if row_site(site#) ≠ 0 then
labell ← row_site(site#)
if cluster_desc(label1) < 0 then
    repeat
        label1 ← − cluster_desc(label1)
        until cluster_desc(label1) > 0
    endif

Update first link of pointer chain to indicate true label –
    cluster_desc(row_site(site#)) ← − cluster_desc(label1)

Add site to existing cluster, merging clusters if necessary –
    if label2 = 0 | label1 = label2 then
        ++ cluster_desc(label1)
        row_site(site#) ← label1
    elseif label1 = 0 then
        ++ cluster_desc(label2)
        row_site(site#) ← label2
    else
        cluster_desc(label1) + cluster_desc(label2) + 1
        cluster_desc(label2) ← − label1
        row_site(site#) ← label1
    endif
    endif
endif
endfor
endproc

Further variables:

scale_factor – The value by which the probability \( p \) is multiplied before approximating it by an integer.

occupation_probability – The scaled integer value corresponding to \( p \).

cluster_desc – The array containing both cluster sizes (positive values) and links between different alias labels (negative values).

label1 – Label of site in previous row that will be overwritten by site of present row.

label2 – Label of preceding site in present row; it is known to be a true label.

7.3.3. Label compression

The next procedure handles the compression of labels and collects completed clusters at the same time.

procedure compress_labels:
    Scan slab perimeter for active clusters and relabel sites with true labels –
    cluster_flag(∗) ← 0
for boundary# = 1 to 4 do
    site_max ← slab_edge
    if boundary# = 2 | boundary# = 4 then site_max ← row#
    for site# = 1 to site_max do
        label ← perimeter_site(site#, boundary#)
        if label ≠ 0 then
            Get true label –
            if cluster_desc(label) < 0 then
                repeat
                    label ← −cluster_desc(label)
                    until cluster_desc(label) > 0
                perimeter_site(site#, boundary#) ← label
            endif
            cluster_flag(label) ← 1
        endif
    endfor
endfor
Scan all true labels –
    active# ← 0
    for cluster# = 1 to cluster_count & cluster_desc(cluster#) > 0 do
        Summarize properties of complete cluster –
        if cluster_flag(cluster#) = 0 then
            update_histogram
            cluster_desc(cluster#) ← 0
        else
            ++ active#
            cluster_desc(active#) ← cluster_desc(cluster#)
            new_label(cluster#) ← active#
        endif
    endfor
    cluster_count ← active#
Compress labels of still-active clusters –
else
    ++ active#
    cluster_desc(active#) ← cluster_desc(cluster#)
    new_label(cluster#) ← active#
endif
endfor
cluster_count ← active#
Relabel perimeter sites with newly compressed true labels –
    new_label(0) ← 0
    for boundary# = 1 to 4 do
        site_max ← slab_edge
        if boundary# = 2 | boundary# = 4 then site_max ← row#
        for site# = 1 to site_max do
            perimeter_site(site#, boundary#) ←
            new_label(perimeter_site(site#, boundary#))
        endfor
    endfor
endproc
Variables:

cluster_flag  – Indicates whether or not cluster extends to perimeter of portion of slab treated so far.
site_max  – Length of current perimeter edge.
active#  – Running count of clusters still active (i.e., still growing).
new_label  – Gives new identity of clusters after label compression (zeroth entry is used for convenience).

7.3.4. Postprocessing

The task of the next procedure of this algorithm, “update_histogram”, is to add the size of a completed cluster to the size histogram. Bins of this histogram cover successive ‘binades’ of the cluster size range; thus the nth bin accumulates the count of clusters with sizes in the range \((2^{n-1}, 2^n - 1)\). The details of the procedure should need no further elaboration.

The final procedure “store_slab” files away the details needed in order to carry out the later task of joining slabs together. The information which must be retained consists of the final state of the slab perimeter together with the cluster-size histogram. Further details as to how the data storage should be organized and how the potentially large number of data files produced in the course of the calculation should be handled are very much a function of the particular computer system on which the work is carried out, and the specifics are inappropriate to this discussion. The description of the algorithm for joining slabs (below) deals with how the information filed away is to be utilized.

The slab production could be concluded with an imposition of open or periodic boundary conditions, and the computation of the final histogram for this size of lattice. Open boundaries amount to simplify truncating all clusters at the periphery, a straightforward operation. Periodic boundaries require that opposite edges of the slab be made adjacent, an operation very similar to that of joining clusters (see below).

7.4. Assembling the lattice

7.4.1. Organizational details

If the slabs are each of size \(L \times L\) then four can be combined to yield a \(2L \times 2L\) lattice, sixteen into a \(4L \times 4L\) lattice, and so on. An alternative way of looking at the combination process is to start with a set of \(4^M\) slabs and combine them in quartets to form \(4^{M-1}\) lattices of edge \(2L\). The next stage of combination is to take these lattices – called regions in what follows – and create \(4^{M-2}\) lattices with edge \(4L\). The combination process continues until just a single lattice embodying all the slabs is produced. All the intermediate lattices are finished off with open or periodic boundary conditions and the cluster distribution data they provide used in the analysis of the functional form of the size dependence.

Each iteration of the assembly process consists, therefore, of joining four selected regions – which may correspond to the original slabs or may be the result of prior combination – into a new region whose number of sites is quadrupled. The algorithm described here is intended for this purpose. For convenience the four regions input to the computation will be referred to as
In terms of compass directions the four quadrants can be identified in the order processed: SW, SE, NE and NW. The first pair of quadrant edges that have to be rendered adjacent consists of the E edge of quadrant 1 (SW) and the W edge of quadrant 2 (SE), or in abbreviated form 1E and 2W; this is followed by the pairs 2N–3S, 3W–4E and 1N–4S. These combination rules apply to the square lattice; for the triangular lattice (for example) there is also the need to carry out a join operation between a pair of diagonally opposite quadrants. How this is done is left as an exercise. The edges of the output region are constructed from pairs of edges from the input quadrants not used so far. The S edge output uses the S edges of quadrants 1 and 2, i.e., S(1 + 2); similarly E(2 + 3), N(3 + 4) and W(4 + 1). All this information regarding the eventual destination of the edge data can be tabulated in order to simplify the decisions that need to be made by the algorithm.

Another difficulty that may present itself when dealing with the larger-sized regions is the lack of processor storage is hold all the required region edge data at once, or, in extreme cases, even just a single complete edge. The edges will then have to be treated in sections, thus further complicating the data handling task. Suffice it to say that the version of the algorithm used in the actual calculations included this feature; the data for each region were stored in five separate files, one for each of the four edges and the fifth for the histogram. All file operations were automated and the file names were constructed so as to include specification of the region size and sequence number as well as the kind of data to be found in the file. We will not dwell on the data handling issue further and will simply assume that the data is made available as needed.

At this stage of the calculation the appearance of very large clusters can no longer be prevented. Those clusters that are too large to be assigned integer sizes, henceforth called 'big' clusters, use floating-point values of sufficient (typically double) precision to represent the sizes exactly. Since the majority of clusters do not fall into this category only minimal use is made of the floating-point capability; special pointers are used to indicate which type of value is intended, as will become apparent from the algorithm itself. The input quadrant files are assumed to include data from these big clusters; since no explicit mention of them was made in the slab algorithm it must be assumed that the storage routine recorded the fact that no such clusters were present. (Should one have the good fortune to use a computer with an adequate integer dynamic range then these considerations become irrelevant.)

7.4.2. Assembly algorithm

This algorithm assembles the data from four quadrants into a single lattice. Details relating to input and output are omitted; the necessary operations are not particularly complicated, but how they might be implemented is closely tied to the computer used.

Accumulate data on incomplete clusters for all four quadrants –

\[
\begin{align*}
\text{cluster\_count}(0) & \leftarrow 0 \\
\text{big\_count}(0) & \leftarrow 0 \\
\text{bin}(\ast) & \leftarrow 0
\end{align*}
\]

Consider each input quadrant in turn –

for quad1# = 1 to 4 do

\[
\text{cluster\_count(quad1\#)} \leftarrow \text{cluster\_count(quad1\#} - 1) + \text{quad\_cluster\_count}
\]
big_count(quad1#) ← big_count(quad1# − 1) + quad_big_count
bin(*) ← bin(*) + quad_big_bin(*)

Concatenate lists of big-cluster sizes –
for count# = 1 to quad_big_count do
    big_size(big_count(quad1# − 1) + count#) ← quad_big_size(count#)
endfor

Concatenate cluster descriptor lists (containing either actual cluster sizes, or pointers
to list of big-cluster sizes which must be adjusted) –
for count# = 1 to quad_cluster_count do
    if quad_cluster_desc(count#) < 0 then
        quad_cluster_desc(count#) ← quad_cluster_desc(count#) − big_count(quad1# − 1)
    else
        quad_cluster_desc(count#) ← quad_cluster_desc(count#)
    endif
endfor

Merge clusters that meet on adjacent quadrant edges –
for quad_neighbor# = 1 to total_neighbors(quad1#) do
    quad2# ← neighbor_table(quad1#, quad_neighbor#)
    for site# = 1 to quad_edge & edge_site2(site#) > 0 do
        Get true labels of sites from both edges –
        label1 ← edge_site1(site#) + cluster_count(quad1# − 1)
        if cluster_desc(label1) + big_offset < 0 then
            repeat
                label1 ← − cluster_desc(label1) − big_offset
            until cluster_desc(label1) + big_offset > 0
        endif
        label2 ← edge_site2(site#) + cluster_count(quad2# − 1)
        if cluster_desc(label2) + big_offset < 0 then
            repeat
                label2 ← − cluster_desc(label2) − big_offset
            until cluster_desc(label2) + big_offset > 0
        endif
        if label1 ≠ label2 then unite_clusters
    endfor
endfor
compress_labels
store_region
complete_lattice
Variables:

- `cluster_count`: Cumulative numbers of cluster as quadrants are added.
- `big_count`: As above, but for 'big' clusters (i.e. those whose size requires a floating-point variable).
- `bin`: Cumulative histogram.
- `quad1#`: The quadrant currently being added.
- `quad_cluster_count`: Contribution to “cluster_count” of each quadrant (†).
- `quad_big_count`: As above, for big clusters (†).
- `quad_bin`: Contribution to histogram of each quadrant (†).
- `big_size`: Cumulative list of big-cluster sizes.
- `quad_big_size`: Contribution of quadrant to “big_size” (†).
- `quad_cluster_desc`: Cluster descriptors for quadrants (†).
- `total_neighbors`: Number of previously treated quadrants (0, 1 or 2) with which the current quadrant shares an edge.
- `quad2#`: A quadrant previously added.
- `neighbor_table`: Identifies the previous quadrants to be considered while adding the current quadrant.
- `quad_edge`: Length of quadrant edge.
- `edge_site1`: A perimeter edge site of the currently treated quadrant (††).
- `edge_site2`: A perimeter edge site of a previously treated quadrant (†, ††).
- `big_offset`: Negative values for “cluster_desc” with absolute value less than “big_offset” are used to designate big clusters; instead of giving the actual cluster size they are interpreted as pointers to the list “big_size” where the actual sizes are held; all cluster label aliases have “big_offset” subtracted from them to avoid ambiguity.

Those quantities marked (†) are input from the stored data (i.e., the files) associated with the current quadrant. Quantities marked (††) are quadrant edges and have the added requirement of being selected according to the orientation (N, E, S, W) of the edge along which the join is being made; the necessary information can be supplied from a table indexed by “quad1#” and “quad_neighbor#” directly to the routine responsible for retrieving the stored data. None of these quantities are indexed explicitly by quadrant number in the algorithm since it is assumed that only the particular version belonging to the currently added quadrant will be available to the algorithm at any instant.

7.4.3. Putting clusters together

This procedure is invoked in order to join two clusters from separate quadrants that are linked across their mutual boundary.

```
procedure unite_clusters:
  Test whether at least one of the clusters is 'big'
  if cluster_desc(label1) < 0 | cluster_desc(label2) < 0 then
```
Get size of 'big' cluster(s) –
  if cluster_desc(label1) < 0 then
    size1 ← big_size(-cluster_desc(label1))
  else
    size1 ← cluster_desc(label1)
  endif
  if cluster_desc(label2) < 0 then
    size2 ← big_size(-cluster-desc(label2))
  else
    size2 ← cluster_desc(label2)
  endif
Save size of combined and adjust pointers to 'big' clusters if needed –
  size ← size1 + size2
  if cluster_size(label2) < 0 then
    big_size(-cluster_desc(label2)) ← size
    if cluster_desc(label1) < 0 then big_size(cluster_desc(label1)) ← 0
    else
      big_size(-cluster_desc(label1)) ← size
      cluster_desc(label2) ← cluster_desc(label1)
    endif
Neither cluster is 'big' –
else
  cluster_desc(label2) ← cluster_desc(label2) + cluster_desc(label1)
Check for possible 'big' combined cluster –
  if cluster_desc(label2) > big_threshold then
    ++ big_count(quad1#)
    big_size(big_count(quad1#)) ← cluster_desc(label2)
    cluster_desc(label2) ← -big_count(quad1#)
  endif
endif
Link clusters –
  cluster_desc(label1) ← -label2 - big_offset
  cluster_desc(edge_site1(site#)) + cluster_count(quad1# - 1) ← ...
  cluster_desc(label1)
endproc

Variable:

  big_threshold – Largest integer-sized cluster allowed.

All the awkwardness of this procedure is a consequence of the use of two representations for cluster size; a mere three lines handle the case where the combined cluster size does not exceed the size threshold. This is just another example of having to dispense with simplicity in order to live with available computer hardware. (The reason for not using just floating-point values for
cluster sizes was to maintain as much commonality between the slab generation and region joining algorithms as possible.)

7.4.4. The remaining computations

The version of the procedure “compress_labels” used for lattice building is functionally equivalent to the version described earlier for slab generation. A total of eight quadrant edges contribute to the perimeter of the newly constructed region (see above). These must be retrieved and used to determine which clusters of the generated region are still active. Completed clusters go to update the histogram. Active clusters are relabeled, as are the sites on the region perimeter. All the added complexity arises from the need to transfer data to and from files and will therefore – as on earlier occasions – not be discussed in detail. Data needed for a further level of region combination is saved by the procedure “store_region” that is an adaptation of “store_slab” mentioned earlier.

The procedure “complete_lattice” finishes the still-active clusters either by truncating them at the perimeter or by wrapping the lattice on a torus to achieve periodic boundaries. The latter case is handled in a similar fashion to the combination of separate quadrants already described. The final histograms used for analysis are produced at this stage.

8. Monte Carlo algorithm for the discrete spin model

8.1. Background

Monte Carlo (MC) techniques have proved of great value in the study of spin models of widely varying kinds [38]. In this final example of algorithm design we show how a new kind of model, designed with the specific intention that it be possible to simulate efficiently, can be studied with the aid of the MC method.

A well-known characteristic of the MC approach is that the greater the number of iterations of the sample-generating process the more reliable (under normal circumstances) the results and the smaller the statistical fluctuations in the sample. The merit of a model for which a faster algorithm can be devised is then all too obvious. For the model described here, all the information that is required during the MC steps is tabulated in advance; the only work that remains to be done by the MC procedure is to manipulate pointers to the tables used. Some early results obtained for the model have been published [39]; while the properties of the basic model are not expected to lead to any surprises, some of the possible extensions (see later) that can be simulated with little extra computational effort are of potential interest.

8.2. The model and its representation

The motivation for introducing a new kind of spin system is the goal of being able to study a relatively complex model while paying the computational cost of a simpler model. The discrete spin model is one in which the spin vectors are allowed to point in a comparatively large number of evenly spaced directions – 30 in fact – which permit it to share some of the behavioral features of the classical Heisenberg model. At the same time the discrete selection of spin
directions removes essentially all the computations required by the usual MC approach to the Heisenberg model [38] leaving a set of manipulations that are only slightly more intricate than for the Ising model.

The energy of the system can be written

$$E(s) = -J \sum_{\langle ij \rangle} s_i \cdot s_j - H \sum_i s_{iz} - U \sum_i s_{iz}^2,$$

where \( \{s_i\} \) is the set of spin vectors; \( J, H \) and \( U \) are the exchange interaction, applied field and uniaxial anisotropy strengths, respectively, the latter two acting on the \( z \)-component of the spins. The spin vectors are of unit length and the first sum is over pairs \( \langle ij \rangle \) of nearest neighbor spins, the others over individual spins. The standard MC procedure as applied to each spin \( s_k \) in turn involves the following steps:

(i) Make a random change to \( s_k \) giving it a new trial direction \( s'_k \).
(ii) Compute the energy difference resulting from the change

$$\Delta E = -J (s'_k - s_k) \cdot \sum_j s_j - H (s'_k - s_k) - U (s'_{kz} - s_{kz})$$

where the sum is over the nearest neighbors of \( s_k \), and then compute

$$\omega = 1/(e^{\Delta E/T} + 1),$$

where \( T \) is the temperature (in suitable units).

(iii) Finally choose a random number \( R \) between zero and unity; if \( R < \omega \) replace \( s_k \) by \( s'_k \), otherwise leave the spin unchanged.

In simulating the Heisenberg model the evaluation of \( \omega \) requires a significant amount of computation, namely the evaluation of the spin products contributing to \( \Delta E \), as well as determining the components of the trial spin \( s'_k \). The advantage gained by restricting \( s_k \) to a finite set of directions is that, as we shall now demonstrate, practically the entire computation can be carried out in advance leaving only some index manipulation to be performed in the course of the simulation itself. In the detailed algorithm we will consider only the case \( H = U = 0 \), but some indication of how the technique can be extended to a wider range of situations will be given.

The number 30 was selected for the allowed number of spin directions to ensure both a reasonably large number of directions (to approximate the continuum) while at the same time preventing the tables that will be introduced shortly from growing excessively. Symmetry requirements also set constraints on the values that are permissible; 30 happens to be the number of edges of both the regular icosahedron and its dual the dodecahedron, and the directions are chosen to be those from the centroid to the midpoints of the edges (of either of the polyhedra). If a \( z \)-direction had to be singled out (e.g., for the external field) it would be chosen to lie along one of these directions. For this particular selection of vectors the values of \( s_k \) are \((0, 0, \pm 1), (\pm \frac{1}{2}, \pm \frac{\tau}{2}, \pm 1/2 \tau)\) and their cyclic permutations, where the choices of sign are independent of one another and \( \tau \) is the ‘golden ratio’ \((\sqrt{5} + 1)/2\) (note that \( 1/\tau = \tau - 1 \)). Subsequent analysis
considers the model for the case of the simple cubic lattice; the table sizes are dependent on the lattice coordination number.

The spin product $s_i \cdot s_j$ can be shown to have 9 possible values $\pm 1, \pm \frac{1}{2}, \pm (\frac{1}{2} + 1/2\tau), \pm 1/2\tau$ and 0. Consequently

$$D = (s_k' - s_k) \cdot \sum s_j$$

$$= n_1(1) + n_2(-1) + n_3(\frac{1}{2}) + n_4(-\frac{1}{2}) + n_5(\frac{1}{2} + 1/2\tau) + n_6(-\frac{1}{2} - 1/2\tau)$$

$$+ n_7(1/2\tau) + n_8(-1/2\tau) + n_9(0).$$  

(4)

where

$$\sum_{i=1}^{9} n_i = 12 \quad (n_i \geq 0)$$  

(5)

since there are 12 spin products that contribute to $D$. Rearrangement yields

$$D = (2n_1 - 2n_2 + n_3 - n_4 + n_5 - n_6)(\frac{1}{2}) + (n_5 - n_6 + n_7 - n_8)(1/2\tau)$$

$$= m_1(\frac{1}{2}) + m_2(1/2\tau),$$  

(6)

where, on account of the restrictions (5), $|m_1| \leq 24$ and $|m_2| \leq 12$. Thus $D$ certainly has no more than 1225 (= 49 * 25) different values assuming the $n_i$ to be independent; since this is not the case the actual number of distinct values of $D$ is less, but it is simpler to ignore this fact in treating the problem.

Returning to (3) and defining $\eta = e^{\ell/T}$ allows us to write

$$\omega = \frac{1}{\eta^{-m_1(1/2) - m_2(1/2\tau)} + 1},$$  

(7)

a quantity which again has only ($\leq$) 1225 values. These values can be tabulated and retrieved with a simple indexing scheme. The table is a set of values $\omega_m$ for $1 \leq m \leq 1225$; the index $m$ is a combination of $m_1$ and $m_2$ appearing in (7) and has the form $m = 25m_1 + m_2 + 613$, a combination which ensures that each valid pair $(m_1, m_2)$ results in a distinct value of $m$. The need to evaluate (3) for each MC step has thus been replaced by a simple reference to a table of precomputed values $\{ \omega_m \}$; what remains to be shown is how to determine the index of the table entry required.

Each MC step involves generating a trial replacement for $s_k$, namely $s_k'$. The new $s_k'$ could be chosen at random from the available spin directions or it could be restricted to one of the four equivalent directions closest to $s_k$. The restricted choice leads to a more gradual change of spin direction as the simulation progresses, although strictly it should not influence the observed equilibrium properties in any way. We therefore will use the restricted selection of $s_k'$; however the alternative unrestricted spin change can be treated by similar means.
Instead of considering the spin vectors $s_k$ directly, we introduce an index variable $g_k$ to specify the direction, where $1 \leq g_k \leq 30$. The indices $g_k'$ associated with the four allowed $s_k'$ that can be reached from $s_k$ are obtained from a $30 \times 4$ matrix $G$–

$$g_k' = G_{g_k t}, \quad (8)$$

where $t$ is a randomly chosen integer between 1 and 4. For the restricted spin changes the terms contributing to (4) can be written

$$(s_k' - s_k) \cdot s_j = \Delta s_k^{(t)} \cdot s_j, \quad 1 \leq t \leq 4. \quad (9)$$

Each such term can result from a total of $4 \times 30 \times 30 = 3600$ possible spin combinations, all of which correspond to unique combinations of the integer quantities $g_k$, $g_k'$ and $g_j$, or equivalently, $g_k$, $t$ and $g_j$. The combined index to the possible values of (9) can be expressed as

$$p^{(t)}_{k j} = 120(g_k - 1) + 30t + g_j; \quad \text{ (10)}$$

the fact that (10) produces a different index for each valid combination of inputs is easily verified.

The next step is to list the actual values of the product (9); despite the 3600 possible ways of selecting the three spin directions there are a mere 9 values that (9) can take, and these are the same as the values for $s_i \cdot s_j$ give previously. Each of these corresponds to a value of $D(6)$, in fact the contribution of each term (9) to the index $m$ needed for referencing $\omega_m$ takes one of the values $\Delta m = \pm 50, \pm 25, \pm 26, \pm 1$ and 0, each corresponding to one of the spin product values listed earlier. If these 9 values of $\Delta m$ are used to fill a 3600-entry table $P(n)$ that is to be accessed via $n = p^{(t)}_{k j}$, then the required value of $\omega$ is $\omega_m$, where

$$m = \sum_{j=1}^{6} P(p^{(t)}_{k j}) + 613. \quad (11)$$

The computations required for each MC step are no more complicated than randomly choosing $t$, evaluating (10) for each neighbor spin and computing the sum in (11), and finally doing the MC test itself to determine whether the spin direction should change. The external field can be included if the indexing scheme is suitably extended; there will then be nine times as many $\omega$ values to be tabulated but otherwise the extra computation amounts to no more than evaluating an additional contribution $\Delta m$ for the field term.

8.3. Pretabulation

The steps shown here produce the tables used in the MC algorithm to be described subsequently. The first step is the derivation of the matrix $G$ that specifies which $g_k'$ correspond to neighboring directions of $g_k$. At the same time, a table of indices, similar to that which will later be stored in $P$, is compiled for the possible spin products $s_i \cdot s_j$. 
neighbor\# \leftarrow 0
index\# = 0
Consider all possible pairs of spin directions –
for direction1\# = 1 to 30 do
    for direction2\# = 1 to 30 do
        spin-product \leftarrow 0
        for component\# = 1 to 3 do
            spin_product \leftarrow spin_product +
            spin(direction1\#, component\#) * spin(direction2\#, component\#)
        endfor
        ... Match product to one of the allowed values and store corresponding index change (note that tests for equality of floating-point quantities should allow some tolerance) –
        count\# \leftarrow 0
        repeat
            ++ count\#
        until spin_product = value_product(count\#)
        ++ index\#
        product_index(index\#) \leftarrow value_index(count\#)
        Store one of the neighbor directions (corresponding to $s_i \cdot s_j = \tau/2$) –
        if count\# = 5 then
            ... neighbor\# = direction2\#
        endif
    endfor
endfor

Variables:

neighbor\#  – Latest entry in table of neighboring directions.
index\#  – Latest entry in table of indices for spin products.
spin_product  – $s_i \cdot s_j$
value_product  – The set of values \{ $\pm 1$, $\pm \frac{1}{2}$, ... \}; the fifth value is $+\tau/2$.
product_index  – Indices corresponding to the possible spin products.
value_index  – Set of possible indices \{ $\pm 50$, $\pm 25$, ... \}.
neighbor_direction  – The stored indices to neighbor directions (matrix $G$).

In the second step the possible index changes ($\Delta m$) are evaluated and stored in table $P$. The table of values of $\omega$ is also filled.

change\# \leftarrow 0
for direction1\# = 0 to 29 do
    for neighbor_choice\# = 1 to 4 do
for direction2# = 1 to 30 do
  + + change#
  index_change(change#) ← ... product_index(30 * neighbor_direction(4 * direction1 # ...
  + neighbor_choice#)) product_index(30 * direction1 # + direction2#)
endfor
endfor
endfor

value# ← 0
for count# = -24 to 24 do
  for count2# = -12 to 12 do
    + + value#
    change_probability(value#) ← ... 1/(exp((− count1 # /2 − count2# /2τ)/temperature) + 1)
  endfor
endfor

Variables:

index_change — Table of indices P.
change_probability — Table of ω values.

The final preparatory step is to set up a matrix that aids in mapping the three-dimensional lattice onto the set of integers (a by-now familiar issue). For interior sites this matrix supplies the offsets to adjacent sites, while for sites on the lattice boundary the offsets provided are to sites on the opposite face that become neighbors when the periodic boundaries are imposed. The matrix is constructed as follows: the lattice is a cube with “edge_length” sites on each edge. Directions +x, −x, +y, −y, +z, −z are labeled 1 through 6; half of the matrix is shown, the other half contains the corresponding negative values.

for site# = 1 to edge_length - 1 do
  offset(site#, 1) ← 1
  offset(site#, 3) ← edge_length
  offset(site#, 5) ← edge_length**2
endfor
offset(edge_length, 1) ← 1 − edge_length
offset(edge_length, 3) ← (1 − edge_length)*edge_length
offset(edge_length, 5) ← (1 − edge_length)*edge_length**2

8.4. The algorithm

The following algorithm carries out a single pass over all sites of the lattice applying the Monte Carlo procedure to the spin on each site in turn.
Loop over lattice sites (the reason a triple loop is used is to provide indices for “offset”) –

- site# ← 0
  
  for site3# = 1 to edge_length do
    for site2# = 1 to edge_length do
      for sitel# = 1 to edge_length do
        ++ site#
      endfor
    endfor
  endfor

Get spins adjacent to current spin –

- spin1 ← spin_direction(site# + offset(sitel#, 1))
  ...
- spin6 ← spin_direction(site# + offset(site3#, 6))

Produce index of trial spin direction –

- trial_index ← 4 * spin_direction (site#) – random_value (1, 4)

Monte Carlo test –

- prob_index ← index_change(30 * trial_index + spin1)
  + ... + index_change(30 * trial_index + spin6) + 613
- if random_value_real (0, 1) ≤ change_probability(prob_index) then
  spin_direction(site#) ← neighbor_direction(trial_index + 1)
endfor
endfor
endfor

Variables:

- spin_direction – The direction index $g_k$; initial values for the simulation may either be chosen at random or all the same.
- trial_index – Index of the trial spin direction (strictly the index - 1).
- prob_index – Used to access the table of $\omega$ values.

The value produced by “random_number_real” is a uniformly distributed floating-point random variate between 0 and 1. The comparatively heavy floating-point computations normally associated with the manipulation of vector spins are obviously absent from what is now a truly simple calculation. (Even the remaining multiplications can be eliminated with a little extra effort.)

8.5. Properties and extensions

The most commonly studied properties of spin systems are the long-range order, the spin pair-correlation, and the fluctuations of these quantities, all as functions of temperature. The long-range order is simply the average spin vector; since in the absence of an external field this quantity, though non-zero for each individual spin configuration below the critical temperature, tends to rotate randomly and thus result in a zero sample average, a physically meaningful measure of the long-range order is therefore based on the absolute magnitude of the spin vector average. Computation of the pair-correlation is a problem closely related to the $\Delta E$ calculation
already described; it too can be reduced to a minimal calculation by the use of a similar indexing scheme and values that are tabulated in advance.

We mentioned earlier how one could go about incorporating a uniform external field into the simulation. The same approach would be used for the anisotropy contribution. Other extensions are equally simple to realize; one in particular is the inclusion of a random field – the field direction is selected at random for each site from among the 30 allowed directions of the spins. An extra array takes care of the field description using a quantity analogous to \( g_k \); the computations are then no more complicated than for the uniform field case.

9. Afterthoughts

The scope of the computations typified by examples given in this paper (‘scope’ typically meaning the size of system studied by Monte Carlo or the number of terms of a series expansion) has tended to grow steadily along with the power of generally available computers. There is little doubt that the trend will continue, perhaps more slowly than previously, but for such purposes the algorithms can be looked upon as recipes and used ‘as is’. Modification, enhancement and wholesale cannibalism will also enable them to be used elsewhere. (The author will be interested to hear how they fare.) More significantly, the case studies should serve an examples of how not too much extra effort can lead to enhanced efficiency and the capability of progressing to larger problems.

To some extent the algorithms are not typical of much of the familiar mainstream of scientific computation – no differential or integral equations are solved, no eigenproblems dealt with, in fact there is practically no floating-point arithmetic at all, with almost all the processing involving only integers. The computations benefit from this fact since most computers still handle integer operations faster than floating-point, although the gap is narrowing. As long as there is a potential gain in using an integer representation of the problem it is well worth the effort to see whether an ostensibly floating-point problem cannot be converted into integer form. Certainly the recent popularity of cellular automata is an indication that the demand for integer computation will persist.

There is a complication facing the future development of algorithms in general and these kinds of algorithms in particular. Given that the computations for the largest problems are extremely time consuming there will be an understandable tendency both to turn to the latest in computer architecture (vector and parallel processors) and to explore the potential of special-purpose hardware (algorithms in silicon, or perhaps gallium-arsenide). The algorithms described here are keyed to the still predominant ‘von Neumann’ serial machine which, at least at the algorithmic level, appears to carry out exactly one operation at a time. Once hardware architecture becomes a factor in algorithm design the algorithms themselves will tend to become more specialized, and software will become even less portable than it is today. But even then, familiarity with how a particular algorithm is constructed for a serial computer may prove a useful departure point for subsequent development.
References