Alternative techniques for cluster labelling on percolation theory

2000 J. Phys. A: Math. Gen. 33 1827

(http://iopscience.iop.org/0305-4470/33/9/308)

J. Phys. A: Math. Gen. **33** (2000) 1827–1840. Printed in the UK PII: S0305-4470(00)02772-4

Alternative techniques for cluster labelling on percolation theory

J Martín-Herrero and J Peón-Fernández

Applied Physics Department, Faculty of Sciences, University of Vigo, 36200 Vigo, Spain

E-mail: julio@uvigo.es and jpeon@uvigo.es

Received 17 March 1999, in final form 5 January 2000

Abstract. We propose a new cluster labelling algorithm as a tool for computer-aided simulation in the field of percolation theory. Due to the use of recursivity, the basic labelling algorithm only needs a few lines of code, and performs at least as well as the Hoshen–Kopelman algorithm with the small lattices that can be stored in the computer memory. Additionally, it can be extended to label the clusters, compute cluster parameters and check for percolation in a single pass over the lattice. We also detail how to deal with lattice dimensions higher than two or with lattices without complete connectivity. Huge lattices, which cannot be stored as a whole in the computer's memory, require a slight modification that permits the labelling by sublattices. In this case, a cluster association technique similar to that proposed by Hoshen and Kopelman is necessary. Nevertheless, the proposed algorithm is capable of labelling lattices of virtually any size, no matter what the memory capacity of the machine, because it does not require an entire line or hyperplane of the whole lattice to be in memory. It is only limited by the storage capacity of the devices attached to the machine. If what is needed is just percolation checking, the storage requirements are very low and processing times decrease, using the 'percolation finders' suggested. Listings of C programs are available in the online edition.

1. Introduction

As for any other cluster labelling algorithm, to simulate a subcrystal of an infinite crystal containing two types (A and B) of randomly distributed molecules, an N-dimensional lattice is generated, assigning to each of its sites one of the types. The sites will form clusters of connected molecules of the same kind. In order to detect in the generated lattice the existence of percolation (the same cluster of type A extending from one side of the lattice to the other), a labelling algorithm is of interest to identify each of the clusters separately, together with other information of interest, such as the average size, maximum size or spatial structure of the clusters. This is a necessary step in order to use group renormalization to obtain the critical exponents for phase transition phenomena (Reynolds *et al* 1980). One such algorithm was proposed by Hoshen and Kopelman (1976; HK hereafter) and since then has been used extensively by the researchers in the field. See, for example, the pioneering book by Stauffer and Aharony (1994) on percolation theory.

The aim of this paper is to give an answer to the challenge that Gould and Tobochnik (1996) made in chapter 12 of their book *Computer Simulation Methods*. The chapter is a nice introduction to percolation theory and group renormalization. They deal with cluster labelling

0305-4470/00/091827+14$30.00

in order to characterize crystals, using the well known Hoshen–Kopelman algorithm. Gould and Tobochnik state, literally,

> although the HK algorithm can be shown to be the most efficient cluster labelling approach to two-dimensional lattices, it is not clear that this approach is the most efficient in higher dimensions. Can you think of a different method for identifying the clusters? (Gould and Tobochnik 1988, p 411.)

We have developed a different collection of techniques for cluster labelling. The algorithm we propose has, at least, the advantage of a greater simplicity in the source code, and, contrasted with the classic HK algorithm, has proven faster when dealing with lattices small enough to be stored in the computer's memory. Nevertheless, due to the great number of implementations, variations and improvements suffered by this well known algorithm, this need not mean that the new algorithm will be faster than any version of the HK algorithm. In fact, we encourage anyone who is working with some version of the HK algorithm to try ours and compare the performance. However, the aim of this paper is not to establish a detailed comparison with the HK algorithm, but to offer the research community in percolation theory an alternative tool that can prove to be useful and more suitable for some tasks than the HK algorithm. Moreover, the new algorithm offers in a single run over the lattice not only the cluster labels, but the determination of percolation (in either direction) together with any of many possible cluster parameters that may be of use, such as dimensions, area, momentum or concentration measurements, for instance. Such information can be computed using a recent version of the HK algorithm, too (Hoshen *et al* 1997). N-dimensional ($N > 2$) lattices are not a problem, either, implying only a few more lines of code.

Really big lattices may either not fit in the computer memory or, if they do fit, can produce a stack overflow error due to the technique being recursive. In these cases it is possible, as explained below, to modify the algorithm to run using sublattices. This implies the introduction of a cluster association technique similar to HK's one, but as little memory demanding as desired. Working with sublattices as small as necessary permits one to deal with lattices as huge as allowed by the storage capacity of the computer, no matter what its memory size or processor speed. What is more, if percolation checking is the only purpose of the labelling, the labelled lattice does not have to be stored and really huge lattices can be computed with the 'percolation finder' routines (programs 5 and 6). Using them, for instance, we have generated and checked for percolation an $L \times L$ lattice with $L = 50\,000$ (2.5×10^9 sites) with a desktop PC (iPentium-II 200 MHz) using less than 3 Mbyte (just storing the whole lattice would need about 10 Gbyte) of disk space, in less than 3 h. Of course, the greater the storage capacity, the greater the lattices that can be simulated. The ratio of consumption of resources versus lattice size (L^2) is quite linear with this algorithm, so it is not difficult to imagine where the limit in lattice size is for a modern desktop PC. We think it is a very acceptable one.

As in any other algorithm for cluster labelling, first of all, a lattice with the desired occupation density has to be generated. The approach used is the same as in the HK algorithm: pseudo-random number generation. By generating a pseudo-random number between 0 and 1 for each of the sites, the site is assigned a -1 if the number is equal to or less than the occupation density and a 0 if it is not. Therefore, sites labelled with -1 are the occupied ones and those with 0 are the empty ones. Sites are filled with molecules in consecutive order. Because of the randomness of the assigning method, it is probable that the occupation density obtained will not be exactly the desired one; i.e. the smaller the lattice, the larger the error. If this is important, it can be useful to select an accuracy bound and to repeat the process until the occupation density lies within the bounds.

Our C programs 1–4, 5.1 and 5.2 are available in, and may be downloaded from, the online edition (`www.iop.org`).

2. The basic algorithm: having it easy with small lattices

Once the matrix $site_{i,j}$ that represents the lattice has been generated, the clusters are labelled to characterize them and to establish the existence (or not) of percolation. The algorithm described here is a single-pass method that requires all of the lattice region to be labelled to be in the computer's memory. In its simplest form, the algorithm is composed of a few lines of code in the main program and a small subroutine, as sketched below in pseudocode.

```
MAIN
    set Label = 0
    for all i, j
        if site_{i,j} = -1
            add 1 to Label
            Label_Site(i, j)
        end of if
    end of for
end of MAIN

LABEL_SITE(i, j)
    set site_{i,j} = Label
    for each neighbour site_{m,n}
        if site_{m,n} = -1 do Label_Site(m,n)
    end of for
end of LABEL_SITE
```

It goes sequentially over all the sites of the lattice until it finds an occupied one. Then it calls the labelling routine (LABEL_SITE) which assigns the current label to the site and searches for occupied sites in the neighbourhood. If it finds one, it calls itself to label the new site with the same label and to search for its neighbours. If not, the routine goes on searching all the neighbouring sites. If there are no occupied sites in the neighbourhood, the routine reaches the end and goes back to the code that called it. If it was the routine itself, running for another site, it will continue searching the rest of the neighbours of that site. If it was called from the main code (the first site of the cluster), it means that all the occupied sites connected with it, i.e. the whole cluster, have been labelled. Then it returns to the main code in search of the next occupied site. Once found, it increases the label number and begins the labelling of the new cluster. Each time that LABEL_SITE is called from the main code for one occupied (still not labelled) site, the whole cluster to which the site pertains is labelled with the contents of the *Label* variable. Then *Label* is increased and the code looks for the next occupied site. There is no risk of labelling a site belonging to an already labelled cluster, because the algorithm only labels sites marked with -1. At the end of the process, all the occupied sites will have a numerical label indicating their cluster, and all the empty ones will be labelled with 0. Program 1 is a short C program that does this (a direct translation of the pseudocode from above). It uses the routines in program 2 to generate and store the lattice. As can be checked, the time used for lattice generation is far greater than for labelling.

To determine the existence or not of percolation, the simplest way is to search the top and bottom rows of the lattice, looking for sites having the same cluster label in both of the rows. This would mean that the cluster extends from one side of the lattice to the other (spanning cluster). Nevertheless, this search is not necessary if the algorithm is enhanced to obtain the

average and maximum cluster dimensions. If so, longitudinal percolation exists whenever the maximum cluster longitudinal size is equal to the lattice longitudinal dimension, and the same in the other directions. Therefore, with the extended algorithm, percolation detection is automatic. The new version is as shown below (see program 3 for a 'ready to use' C implementation).

```
MAIN
    set Label, Lon_max, Transv_max = 0
    set Lon_ave, Transv_ave = 0
    for all i, j
        if site_i,j = −1
            add 1 to Label
            set Lon, Transv = 0
            set i_min, i_max = i
            set j_min, j_max = j
            Label_Site (i, j)
            set Transv = i_max − i_min + 1
            if Transv_max < Transv set Transv_max = Transv
            add Transv to Transv_ave
            set Lon = j_max − j_min + 1
            if Lon_max < Lon set Lon_max = Lon
            add Lon to Lon_ave
        end of if
    end of for
    divide Lon_ave, Transv_ave by Label
end of MAIN

LABEL_SITE(i, j)
    set site_i,j = Label
    if i_min > i set i_min = i
    if i_max < i set i_max = i
    if j_min > j set j_min = j
    if j_max < j set j_max = j
    for each neighbour site_m,n
        if site_m,n = −1 do Label_Site(m,n)
    end of for
end of LABEL_SITE
```

It has grown a bit in length but not too much in complexity. Once run, it offers in $Lon_{\max}$ and $Transv_{\max}$ the maximum cluster length and width, and the average values in Lon_{ave} and $Transv_{\mathrm{ave}}$. Just storing in imax, imin, jmax and jmin the furthest positions that the subroutine reaches during the labelling of each single cluster does it. Thus it obtains the length and width of the cluster. If necessary, together with the cluster label its width, length or any other cluster structure related parameter that may be useful can be stored, such as momentum, area (number of sites belonging to it), dispersion, concentration measurements, etc, computing them in a similar way. As mentioned, with this version of the algorithm, in a single line of code the program can detect the existence of longitudinal percolation, just by comparing $Lon_{\max}$ with the lattice longitudinal dimension, and using $Transv_{\max}$ for the transverse direction. The spanning clusters in any direction can be identified by means of their dimensions, if stored.

What is more, if the purpose of the labelling is just to detect the existence or not of longitudinal (transverse) percolation, i.e. a 'percolation finder', the code only needs to run for the first row (column) of the lattice, and is stopped as soon as j (i) is equal to the lattice longitudinal (transverse) dimension, i.e. as soon as a bottom site is reached to be labelled,

without the need for any other local variable. This can be seen as pouring water into the lattice through the holes (clusters) in the upper side (first row) and just waiting to see whether some appears at the bottom. It is not necessary to label every cluster in the lattice, just those starting at the first row, and even this process is stopped as soon as the bottom (last row) is reached. Also, the 'water drops' are programmed to 'feel' the gravity and tend to go downwards whenever it is possible. This is a very fast method for percolation detection, much faster than the HK algorithm, merely because it really does a much smaller job (remember that it does not label the whole lattice, it just checks for percolation). One such 'percolation finder' in C is program 4. With this program, percolation checking time is incomparably less than the lattice generation time. We have not tested program 4 versus the HK algorithm because the HK labels the whole lattice prior to the percolation checking and therefore the time consumed is not directly comparable. If what is needed is just percolation checking (in lattices that can be stored in memory as a whole), there should be no doubt regarding its performance and simplicity.

Working with higher-order lattices is easy. It only implies using many lattice dimension indices, $i, j, k, \ldots$ as the order of the lattice, and to increase the neighbourhood search code in the subroutine to include the new possible connections. In a similar way, in a three-dimensional matrix, for instance, the neighbourhood does not necessarily have to be made of all six sites next to the site, as in a cubic lattice. A triangular or any other connection pattern can be defined and implemented easily, just by limiting the search directions, or adding new ones. As the lattice order increases, the search for percolation and the computation of cluster dimensions have to be adapted accordingly.

We have run some simple tests using program 3 (labelling the whole lattice and percolation detection at the same time) versus one of the many available implementations of the HK algorithm (Stauffer and Aharony 1994). We used the same lattice generation routine for both, which is detailed in C code in program 2. Each method was run 100 times for 100 lattices with densities of occupation of 0.15, 0.35, 0.70 and 0.90, and three different sizes. The time elapsed for the generation and labelling of the 100 lattices was averaged along the 100 runs for each density and size. The results can be seen in table 1 (two-dimensional lattices) and table 2 (three-dimensional lattices). There is a column for each occupation density and one row for each lattice size.

Table 1. Time consumed by the HK algorithm and that proposed by the authors in generating and labelling 100 2D lattices with different p (top) and size (left). The time consumed by the authors' algorithm is shown in parentheses.

	0.15	0.35	0.70	0.90
50×50	3 (1)	3 (1)	3 (1)	2 (1)
100×100	7 (4)	7 (4)	6 (4)	6 (4)
500×500	138 (92)	150 (94)	121 (100)	115 (102)

Table 2. Time consumed by the HK algorithm and that proposed by the authors in generating and labelling 100 3D lattices with different p (top) and size (left). The time consumed by the authors' algorithm is in parentheses.

	0.15	0.35	0.70	0.90
$21 \times 21 \times 21$	5 (3)	4 (3)	4 (4)	5 (4)
$25 \times 25 \times 25$	8 (5)	8 (5)	7 (6)	7 (7)
$41 \times 41 \times 41$	33 (25)	36 (25)	32 (27)	32 (28)

The average time consumed for the generation and labelling of 100 lattices is shown in seconds, running on a PC with an iPentium-II 200 MHz processor and 96 Mbyte of RAM. The time consumed by the HK algorithm is shown and within parentheses is that corresponding to our method. It can be seen that the performance of the HK algorithm seems to decrease around $p = 0.35$, while the performance of the proposed method decreases with increasing density of occupation, as should be expected due to there being more sites to label. If we fix our attention on the bigger lattices, we find an average time of the proposed method of 75% of the HK algorithm time for two-dimensional lattices of size 100^2 and 79% for three-dimensional lattices of size 41^3. Additional tests, on the same computer, with 100 runs for 100 lattices each, of size 1000^2, $p = 0.45$, give 537 s (i.e. 5.37 s/lattice) for the HK algorithm and 384 s (3.84 s, 71.5%) for the proposed algorithm. Similar tests for three-dimensional lattices of size 100^3, $p = 0.45$, have given 837 s (i.e. 8.37 s/lattice) for HK and 386 s (3.86 s, 46%) for the proposed method. In addition, the proposed method computed simultaneously the cluster labels, the average and maximum cluster size in every direction, and the existence or not of percolation in every direction.

Nevertheless, these results have to be handled with care because, as mentioned above, many different implementations of the HK algorithm exist and it could happen that they might give significantly different results. However, at least, they should serve to support the new method as a valid alternative, not only for its simplicity (see programs 2 and 3) but for its performance too (not to mention the 'percolation finder', program 4). However, the reader must not forget that these results are for the basic version, which requires the whole lattice to be in the computer memory. Both algorithms were tested working with the whole lattice into the computer memory. The HK algorithm needed additional memory space for the index-of-indices array, which, without the use of recycling techniques, caused some problems for the machine when we tried bigger lattices. With the proposed algorithm similar problems arose, regarding memory capacity, specifically the stack size for recursion, but at higher lattice sizes than with the HK algorithm (usually most compilers permit one to adjust the stack size).

It is easy to see that one important subject regarding this kind of labelling algorithm is the memory stack. It is there where the computer stores the program counter and the local and status variables each time a subroutine is called. Therefore, the number of local variables should be low, on account of possible problems of stack overflow. It is easy to do some calculations: take the case of a two-dimensional square lattice of size 3000^2; one can easily expect, depending on the occupation density, clusters of up to 9×10^6 sites (with a density of occupation equal to 1, not a particularly useful study case, but significant as an upper limit to our calculations). The labelling of such a cluster would involve a recursive call to the subroutine no more than 9×10^6 times (in general, the number of accumulated routine calls would be significantly less than the number of sites in the cluster. The upper limit, equality, is only reached in square clusters and the like). Each routine call has to store in the stack the program counter (a memory address) and some local variables (at least two integers, plus status registers). They remain there until the routine ends its code. Therefore, to prevent a stack overflow error when using the basic algorithm with a 3000^2 lattice, we would need a stack of greater than 100 Mbyte. The alternative is to use the extended version described below.

3. The extended algorithm: towards bigger lattices

Due to the stack being a part of the RAM memory, which has to store the lattice itself, when dealing with big lattices and/or high occupation densities it is necessary to label the lattice using sublattices, which implies a modification of the algorithm. First, the lattice must be divided into overlapping sublattices, with an overlap extension of just one site along each

border. Then the sublattices are sequentially labelled with a modified version of the algorithm. Subdivisions and disk have already been used (Rapaport 1985).

The way it works is essentially the same. There is just a slight difference if it finds a site with a label which is less than the label value with which the current sublattice started. This can only happen on two of the borders of the sublattice, and would mean that the site belongs to a cluster that has already been partially labelled in a previous sublattice. Therefore, the rest of the connected sites have to be labelled with the old label. However, here we can face the same kind of problem as when labelling with the HK algorithm, i.e. the cluster labels assigned in previous sublattices to what seemed to be different clusters may be wrong if they belong to a unique cluster spanning several sublattices. Thus it is necessary to include some way of associating clusters which really are the same but have different labels. For the sake of speed, an array of associated indices can be used, much the same as used in the HK algorithm. However, it has the disadvantage of underusing a lot of array positions, those of the cluster labels that are not associated with any other. Nevertheless, when computing speed prevails over memory usage, this can be a useful method. In pseudocode:

```
MAIN
    set Label = 1
    for all sublattice
        set Label_ini = Label
        for all i, j in sublattice
            if (i = 0 or j = 0) and (0 < site_i,j < Label_ini)
                set Label_real = site_i,j
                Label_Site(i, j)
            else if site_i,j = −1
                set Label_real = Label
                add 1 to Label
                Label_Site(i, j)
            end of if
        end of for
    end of for
    Reset_Labels()
end of MAIN

LABEL_SITE(i, j)
    set site_i,j = Label_real
    for each neighbour site_m,n
        if site_m,n = −1 do Label_Site(m,n)
        else if site_m,n > 0 and <> Label_real do
            Associate_Labels(site_m,n, Label_real)
            Label_Site(m,n)
        end of if
    end of for
end of LABEL_SITE

ASSOCIATE_LABELS(a, b)
    if a < b do Swap(a, b)
    if Assoc[a] <> 0
        if Assoc[a] = b do Exit
        else Associate_Labels (Assoc[a], b)
    else set Assoc[a] = b
end of ASSOCIATE_LABELS
```

```
RESET_LABELS()
    set Label = 1
    for all a
        if Assoc[a] = 0 do
            set Assoc[a] = Label
            add 1 to Label
        else set Assoc[a] = Assoc[Assoc[a]]
    end of for
end of RESET_LABELS
```

After running RESET_LABELS() at the end of the labelling routines, in *Assoc*[*Label*] we will have for each label a real label which identifies all the associated clusters as only one. There will only be one label for each cluster and all the labels will be consecutive, starting from 1. The association process can be seen in table 3. In table 4 we show the resetting process. ASSOCIATE_LABELS and RESET_LABELS are very efficient routines, and both can be used to deal with the array of indices in the HK algorithm, too.

Table 3. Working method of ASSOCIATE_LABELS(a, b). If the label to associate, a, is already associated, it tries to associate the target, b, with the label associated with the first, $A[a]$. And goes on trying until it finds one not yet associated. It always associates the lesser to the greater, to afford computation time in RESET_LABELS() and to prevent endless loops.

	Array position									
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	Action
Array contents										
0	0	0	0	0	0	0	0	0	0	Starting state.
0	0	**2**	0	0	0	0	0	0	0	Associate(3, 2)
0	0	2	0	**2**	0	0	0	0	0	Associate(5, 2)
0	**1**	**2**	0	2	0	0	0	0	0	Associate(1, 3) $\rightarrow$ (3, 1) $\rightarrow$ (2, 1)
0	1	2	0	2	0	0	**4**	0	0	Associate(8, 4)
0	1	2	0	2	0	**5**	4	0	0	Associate(7, 5)
0	1	2	**2**	2	0	5	4	0	0	Associate(4, 2)
3	1	**2**	**2**	2	0	5	**4**	0	0	Associate(8, 3) $\rightarrow$ (4, 3) $\rightarrow$ (2, 3) $\rightarrow$ (3, 2) $\rightarrow$ no action
0	**1**	2	2	**2**	0	5	4	0	0	Associate(5, 1) $\rightarrow$ (2, 1) $\rightarrow$ no action

The sublattices can be, for example, blocks of 100×100 sites, the first from $(0, 0)$ to $(100, 100)$, the second from $(100, 0)$ to $(200, 100)$, and so on, or whatever value is the best suited for the characteristics of the computer, never forgetting the overlap between adjacent sublattices. A good way to find an upper limit for the blocking factor (sublattice size) is to use the worst case of stack use (one cluster over the whole sublattice) to know the memory required. It would be the size of the sublattice plus the maximum stack size plus a security margin for the other variables and operating system requirements. The maximum stack size needed for the recursion can be estimated as the sublattice size times the size of a memory address (program counter) plus the lattice dimension times the size of an integer (site position).

This version permits one to have in memory just one sublattice at a time, while the entire lattice remains in a massive storage device. Thus the computer memory is not a limit for the size of the lattice. This is only limited by the size of the massive storage device(s) the computer has access to. With a sublattice size as mentioned above of 101^2, for instance, it would require only about 240 Kbyte of RAM memory (taking 32 bits per label), and there would be no limit in the number of sublattices, permitting one to reach virtually infinite dimension lattice, because the size of the sublattices to use does not depend on the lattice size. In that way this is an algorithm which is not very demanding for small memory machines but is still fast and very

Table 4. Working method of RESET_LABELS(). If the current label, a, has any associate, b, it will be less than the current label, a (because of ASSOCIATE_LABEL) and thus it will have already correctly associated a label, $A[b]$, so it is directly usable for the current label, $A[a] = A[b]$. If not, use the value stored in *Label*, $A[a] = Label$, to ensure correlativity.

Array position										
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	Action
Array contents										
0	1	2	2	2	0	5	4	7	6	Starting state ($Label := 1$)
1	1	2	2	2	0	5	4	7	6	$A[1] = 0$ $A[1] := Label = 1$ ($Label := 2$)
1	**1**	2	2	2	0	5	4	7	6	$A[2] <> 0 \rightarrow A[2] := A[A[2] = 2] = 1$
1	**1**	**1**	2	2	0	5	4	7	6	$A[3] <> 0 \rightarrow A[3] := A[A[3] = 2] = 1$
1	**1**	1	**1**	2	0	5	4	7	6	$A[4] <> 0 \rightarrow A[4] := A[A[4] = 2] = 1$
1	**1**	1	1	**1**	0	5	4	7	6	$A[5] <> 0 \rightarrow A[5] := A[A[5] = 2] = 1$
1	1	1	1	1	2	5	4	7	6	$A[6] = 0 \rightarrow A[6] := Label = 2$ ($Label := 3$)
1	1	1	1	**1**	2	**1**	4	7	6	$A[7] <> 0 \rightarrow A[7] := A[A[7] = 5] = 1$
1	1	1	**1**	1	2	1	**1**	7	6	$A[8] <> 0 \rightarrow A[8] := A[A[8] = 4] = 1$
1	1	1	1	1	2	**1**	1	**1**	6	$A[9] <> 0 \rightarrow A[9] := A[A[9] = 7] = 1$
1	1	1	1	1	**2**	1	1	1	**2**	$A[10] <> 0 \rightarrow A[10] := A[A[10] = 6] = 2$

simple. With respect to storage capacity, if again only a 'percolation finder' is needed, only one row of the lattice would have to be stored at a time: the last row of every sublattice, to be read at the time of labelling the adjacent sublattice. It would not be necessary to store the last column of each sublattice because subsequent sublattices can be labelled consecutively.

If more than a 'percolation finder' is wanted, and therefore the whole sublattices have to be stored, it is advisable to store them upside down or to label the lattice starting from bottom to top, just to make it possible that the unique row that has to be read from each sublattice for the labelling of its adjacent sublattice is the first row in the file, which improves the reading speed. However, for percolation detection in huge lattices, it would suffice if there were room enough in the storage devices for one row of the lattice (to be used only every n rows, n being the number of rows in a sublattice), a list of the labels used in the first row of the lattice, another one for the labels in the last row of the lattice, and the array of associated labels (*Assoc*[*Label*]) to look for coincidences in both rows.

If we are dealing with very big lattices, we face the problem of a very big array of association of labels. Let us take the case of a $2\,000\,000^2$ lattice. We would only need (32 bits per site) 8 Mbyte for the only row to store, and 4 Mbyte (10^6 different clusters as the maximum in a row) for each list of labels (first and last lattice rows). This is 16 Mbyte of storage capacity for a $2\,000\,000^2$ lattice, no matter how much (or how little) RAM memory: the smaller the memory, the smaller the sublattices. However, there is an array of associations: we should expect an upper limit (reachable only for very low occupation densities) for the number of labels of about the occupation density times the size of the lattice. So we should need an association array of about $4 \times 2^2 \times 10^6 \times p$ Mbyte. This is a lot of storage capacity. That is why it is necessary to think about some other way to manage the cluster associations.

4. One last modification: huge lattices within reach

The modification we are going to introduce is another way of managing the association of clusters between sublattices. The main problem with *Assoc*[*Label*] is the necessity of having

an array index for every cluster label, whether it is associated with any other or not. If we think about the worst possible case, the maximum number of clusters we can have in a sublattice happens when the sublattice is similar to a chessboard, that is, one-half of the total number of sites in the sublattice. Therefore, in the whole lattice, we would have as many possible clusters as the number of sublattices times one-half the number of sites per sublattice, i.e. one-half the size of the sublattice. That would be 10^{12} possible clusters for a lattice of size of $2\,000\,000^2$. If we now consider only those clusters which can be associated with others, i.e. only the maximum possible number of associations between clusters which we can find, we find that we can only have cluster associations on two of the edges of the sublattices, and as many different clusters as we could find there, that is (again considering 'chessboard-like' sublattices) half the number of sites in one edge plus half the number of sites on the other. In square lattices of size L^2 this would be equal to L maximum possible associations per sublattice. If we again take the $2\,000\,000^2$ lattice, and divide it into 2000×2000 sublattices of size 1000^2, we would have an (unreachable) maximum number of associations of 4×10^9. That is, we would be using only one of every 250 positions in *Assoc*[*Label*]. If we were to use 200×200 sublattices of size $10\,000^2$, then the array usage would be only one of every 2500 positions in the array, and it would be enough with 1.6 Gbyte to store the associations for a $2\,000\,000^2$ lattice (far enough, because this upper limit is really high!).

Therefore, the necessity for some list of associations where only the clusters associated with others occupy memory is obvious. The solution is a trade-off of storage capacity savings in exchange for processing speed. Two arrays, $A[c]$ and $B[c]$, will be used to store the labels of associated clusters. Two labels stored at the same position in $A[c]$ and $B[c]$ (same c) will be associated. Labels in $A[c]$ will always be greater than the corresponding labels in $B[c]$. With this strategy, the rest of the algorithm remaining the same, it is enough to change ASSOCIATE_LABELS and RESET_LABELS for those that follow.

```
ASSOCIATE_LABELS(a, b)
    if a < b do Swap(a, b)
    set c = 0
    while A[c] <> a and c < cmax do add 1 to c
    if A[c] = a
        if B[c] = b do Exit
        else Associate_Labels (B[c], b)
    else
        set A[c] = a
        set B[c] = b
        add 1 to cmax
    end of if
end of ASSOCIATE_LABELS

RESET_LABELS()
    sort A[c], B[c] using A[c]
    for all c < cmax
        search for d < c such that A[d] = B[c]
        if exists set B[c] = B[d]
    end of for
end of RESET_LABELS
```

The number of associations stored in $A[c]$ and $B[c]$ is stored in $c_{\max}$. To improve the speed in RESET_LABELS when dealing with big values of $c_{\max}$, $A[c]$ and $B[c]$ are sorted using some efficient sorting algorithm (such as Quicksort or Heapsort, we recommend the latter due to memory savings. See, for example, the masterpiece by Press *et al* (1992)).

Table 5. Working method of ASSOCIATE_LABELS(a, b). If the label to associate, a, is already associated, it tries to associate the target, b, to the label associated with the first, $A[c]$. It goes on trying until it finds one not yet associated. It always associates the lesser to the greater, to afford computation time in RESET_LABELS() and to prevent endless loops.

c	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	Action
A	0	0	0	0	0	0	0	0	0	0	Starting state
B	0	0	0	0	0	0	0	0	0	0	
A	**3**	0	0	0	0	0	0	0	0	0	Associate(3, 2) ($c_{\max} = 0$)
B	**2**	0	0	0	0	0	0	0	0	0	
A	3	**5**	0	0	0	0	0	0	0	0	Associate(5, 2) ($c_{\max} = 1$)
B	2	**2**	0	0	0	0	0	0	0	0	
A	**3**	5	**2**	0	0	0	0	0	0	0	Associate(1, 3) $\rightarrow$ (3, 1) $\rightarrow$ (2, 1) ($c_{\max} = 2$)
B	**2**	2	**1**	0	0	0	0	0	0	0	
A	3	5	2	**4**	0	0	0	0	0	0	Associate(4, 2) ($c_{\max} = 3$)
B	2	2	1	**2**	0	0	0	0	0	0	

Table 6. Working method of RESET_LABELS(). If the label associated, b, to the current label, a, already has an associate, it will be less than the current label, a (because of ASSOCIATE_LABEL and the previous sorting of A and B) and thus it will exist $d < c$ so that $B[d]$ is the correct label, $B[c]$, for a. If not, $B[c]$ is the minimum of the labels associated with $A[c]$, therefore remains untouched, and will be used for the rest of the labels associated with $A[c]$ in due time.

c	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	Action
A	3	5	2	4	8	9	0	0	0	0	Starting state ($c_{\max} = 6$)
B	2	2	1	2	4	6	0	0	0	0	
A	**2**	**3**	**4**	**5**	**8**	**9**	0	0	0	0	Sort
B	**1**	**2**	**2**	**2**	**4**	**6**	0	0	0	0	
A	2	3	4	5	8	9	0	0	0	0	$c = 0$ (there is no $d < c$)
B	**1**	2	2	2	4	6	0	0	0	0	
A	2	3	4	5	8	9	0	0	0	0	$c = 1$ ($d = 0$)
B	**1**	**1**	2	2	4	6	0	0	0	0	
A	2	3	4	5	8	9	0	0	0	0	$c = 2$ ($d = 0$)
B	**1**	1	**1**	2	4	6	0	0	0	0	
A	2	3	4	5	8	9	0	0	0	0	$c = 3$ ($d = 0$)
B	**1**	1	1	**1**	4	6	0	0	0	0	
A	2	3	4	5	8	9	0	0	0	0	$c = 4$ ($d = 2$)
B	1	1	**1**	1	**1**	6	0	0	0	0	
A	2	3	4	5	7	9	0	0	0	0	$c = 5$ (there is no $d < c$)
B	1	1	1	1	1	**6**	0	0	0	0	

Both are sorted depending on the contents of $A[c]$. The recommended searching strategy for d in RESET_LABELS is to test recursively the midpoint of the adequate side of the segment to scan (see program 5.2). The way these routines work can be seen in table 5 and 6.

With these versions of ASSOCIATE_LABELS and RESET_LABELS, there is no problem at all in using a different pair of arrays, $A[c]$ and $B[c]$, for each sublattice. Thus, only a maximum array size L (far less would generally suffice) would be needed for a sublattice of size L^2, to avoid out of memory errors during the labelling process. After each sublattice processing, both arrays can be dumped to a file. At the end of the last sublattice, it would suffice

to read that file using ASSOCIATE_LABELS (to avoid duplicates and make the due connections between separate sublattices) and then processing the entire arrays with RESET_LABELS.

Programs 5.1 and 5.2 use these techniques to check for percolation in big lattices. Program 5.1 performs the labelling and program 5.2 does the resetting and relabelling of the stored rows. We do it with two separate programs for better memory management (otherwise, we would have to reserve more memory for the arrays A and B during the labelling, and therefore we would have to use smaller sublattices). The alternative is to use dynamic memory allocation to increase the memory allocated for the arrays A and B and to free the memory allocated for the sublattice prior to the resetting and relabelling phase. However, these techniques, really advisable, could lead to a loss in clarity and therefore have not been used in any program in this paper.

The sublattices are generated just before their labelling, and only the first row is dumped to disk after the labelling. As the lattice is processed from bottom to top, it is only necessary to store the first row for each sublattice, and only until the next one above it is processed. The last row of the bottom sublattices is stored, too, to perform the percolation checking (of course it is possible and easy to adapt the code in programs 5.1 and 5.2 to store the whole sublattices, if the whole labelled lattice is needed, but it would require much more disk space). The association arrays, $A[c]$ and $B[c]$, of each sublattice are appended to the file 'assoc.txt'. When the labelling of all the lattices is finished, program 5.2 is used to read the file 'assoc.txt' and to reset the associations as mentioned above. Then the last row of the lattice and the first row of the top sublattices (top row of the lattice) are relabelled according to the resulting arrays of associations. To finish, it just remains to do a half-minute test to search for coincidences between both rows (not included in program 5.2). For clarity and the editor's economy, program 5.2 uses a very short but very inefficient sorting routine. It is strongly recommended to change it to HeapSort if really big association arrays are going to be processed. The results that follow were all obtained using programs 5.1 and 5.2 as they are, without any of the suggested modifications.

With those 'percolation finders' 100 lattices of size 5000^2, using 10×10 sublattices of size 500^2, with $p = 0.45$, have been simulated on the same computer as above. The test averaged 102 s per lattice (the coincidence check between rows using a non-optimized routine took less than 1 s) and needed on average 108 Kbyte of disk space. Testing 10 lattices of size $20\,000^2$ (using 20×20 sublattices of size 1000^2), $p = 0.45$, averaged 1638 s (about 28 min) per lattice, with a mean disk usage of 823 Kbyte. And lastly, using the same desktop PC, generation and percolation checking of one lattice of size $50\,000^2$ (using 25×25 sublattices of size 2000^2), $p = 0.45$, took 10 321 s (2 h and 52 min) and 2.67 Mbyte of disk space. The detailed results can be seen in table 7.

In table 7 two additional columns have been included that deserve further explanation. The factors that lead from each variable in the previous column to the next have been specified. Thus we can have a hint about the algorithm's performance linearity with respect to lattice characteristics. We can see that the total computing time increases linearly with the lattice size (at least as far as the generation and labelling time being the greatest part of it). Disk usage grows linearly with the number of associations, which itself grows linearly with the relation lattice size factor to sublattice length factor ($\frac{16}{2}$ and $\frac{6}{2}$, respectively), accordingly with what was mentioned above when estimating the number of possible associations. This holds whenever disk usage is dominated by the associations file. Obviously, for the first and last rows disk usage varies linearly with the row size ($\times 4$ and $\times 2.5$, respectively). Association resetting time is the only parameter that does not behave linearly due to the nonlinearity of the techniques involved. Lastly, the density of occupation must not be forgotten. All results in table 7 refer to $p = 0.45$.

Table 7. Testing programs 5.1 and 5.2, the 'percolation finders' for big lattices.

Lattice size (sites)	2.5×10^6	$\times 16$	4×10^8	$\times 6.25$	2.5×10^9
Number of sublattices	10×10	$\times 4$	20×20	$\times 3.1$	25×25
Sublattice length	500	$\times 2$	1 000	$\times 2$	2 000
Total time (s)	102	$\times 16$	1 638	$\times 6.3$	10 321
Generation and labelling (s)	99	$\times 16$	1 578	$\times 6.2$	9 767
Associations resetting (s)	2	$\times 28$	57	$\times 9.7$	550
Rows relabelling (s)	1		3		4
Total disk usage (Kb)	108	$\times 8$	823	$\times 3.2$	2 670
Associations file (Kb)	67	$\times 10$	653	$\times 3.4$	2 190
First and last row (Kb)	41	$\times 4$	170	$\times 2.9$	494
Number of associations	4434	$\times 8$	37 198	$\times 3$	114 865

Therefore, it is easy to use some tests with medium-sized lattices to predict where the limit for huge lattice simulations is using a given machine, and then proceed with any desired lattice within that machine's reach.

One last word about these results. For the 2500×10^6 lattice, about 222×10^6 labels were needed. Of those, only 114 865 were associated (bad ones, in terms of the HK algorithm) to any other. This represents about one associated label out of every 2000 labels, i.e. the 0.0005%.

5. One last remark

As can be concluded from what has been said and seen, simulating much bigger lattices (could we call them huge lattices?) in search of percolation is possible using just a desktop PC with enough free disk space and a normal amount of RAM, in reasonable lapses of time (not to mention using more powerful machines: exactly the same test as in the previous section, with the $50\,000^2$ lattice, took only 25 min on a desktop iPentium III 450 MHz PC). We are not saying that this cannot be achieved (with such modest hardware requirements) with the HK algorithm, but there are several things to consider: first, it is probable that the labelling of every sublattice will be faster with the proposed algorithm than with the HK. Secondly, there is no lower limit in the sublattice size, so there are no minimum requirements concerning the computer's memory. Thirdly, the number of cluster associations (bad labels, in HK terms) will always be much less with the proposed algorithm than with the HK (a mere question of scale: the HK algorithm accumulates all the possible associations due to the entire range of sublattice sizes; the proposed algorithm only accounts for the associations due to one chosen sublattice size. Our algorithm could be thought of as a modified HK algorithm running for sublattices instead lattice sites).

Still, again the authors want to remark that their intention is to open a new door for those engaged in computer-aided simulation in the field of percolation theory, and contribute to making complex simulations easier and more accessible to those who do not have expensive hardware at their disposal. This different approach to cluster labelling tries to be another tool in the hand of researchers. Now they have the last word about its opportunity, utility and possibilities for the future.

References

Gould H and Tobochnik S 1996 *An Introduction to Computer Simulation Methods: Applications to Physical Systems* vol 2, 2nd edn (New York: Addison-Wesley)

Hoshen J, Berry M W and Minser K S 1997 Percolation and cluster structure parameters: the enhanced Hoshen–Kopelman algorithm *Phys. Rev.* E **56** 1455

Hoshen J and Kopelman R 1976 Percolation and cluster distribution. Cluster multiple labelling technique and critical concentration algorithm *Phys. Rev.* B **14** 3438

Press W H, Teukolsky S A, Vetterling W T and Flannery B P 1992 *Numerical Recipes in C* 2nd edn (Cambridge: Cambridge University Press) pp 332–8

Rapaport D C 1985 *J. Phys. A: Math. Gen.* **18** L175

Reynolds D J, Stanley H E and Klein W 1980 Large cell Monte Carlo renormalization group for percolation *Phys. Rev.* B **21** 1223

Stauffer D and Aharony A 1994 *Introduction to Percolation Theory* (London: Taylor and Francis) A FORTRAN implementation of the H-K algorithm is given in the appendix