Available online at www.sciencedirect.com

SCIENCE DIRECT®

Journal of Computational Physics 206 (2005) 578–596

JOURNAL OF COMPUTATIONAL PHYSICS

www.elsevier.com/locate/jcp

About two cluster generating algorithms

Gelu M. Nita *,1

Department of Physics, New Jersey Institute of Technology, University Heights, Newark, NJ 07102-1982, USA

Received 5 July 2004; received in revised form 3 January 2005; accepted 4 January 2005
Available online 8 February 2005

Abstract

Two original algorithms used to generate *n*-element clusters in a rectangular lattice are presented. The first algorithm, suitable for simulations related to thermodynamic investigations of metastable cluster systems [V.A. Shneidman, G.M. Nita, On the critical cluster in the 2-dimensional Ising model: computer-assisted exact results, J. Chem. Phys. 121 (11232) 2004], is able to generate all distinguishable cluster configurations with a given number of elements. The maximum number of elements forming such two dimensional configurations is limited only by the computational time involved, which is shown to be essentially improved by using a parallel processing approach. The complete results up to 17 cluster elements are presented. The second algorithm, suitable for dynamic simulations [V.A. Shneidman, G.M. Nita, Modulation of the nucleation rate preexponential in a low-temperature Ising system, Phys. Rev. Lett. 89 (2002) 025701; Phys. Rev. E, 68 (2003) 021605], is able to generate full or partial nucleation chains up to a maximum cluster size limited by the computational time and the virtual memory available. For illustration purposes, the complete nucleation chains up to a 5 elements and partial nucleation chains up to 13 elements are presented in detail, although complete results up to 10 elements and partial nucleation chains up to 100 elements were successfully produced using this algorithm.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Cluster configurations; Ising model

* Tel.: +1 973 596 3437; fax: +1 973 596 3617.
E-mail address: gnita@njit.edu.
URL: http://web.njit.edu/gnita/.

[1] The author wants to thank Vitaly Shneidman for initiating this project, for his advice during the elaboration of these algorithms and for encouraging their stand-alone publication.

doi:10.1016/j.jcp.2005.01.001

1. Introduction

The problem of finding the exact number of distinguishable cluster configurations possible to be formed, provided a lattice geometry, with a given number of elements has always been of particular interest due to its applications related to the nucleation theory (see for more details [5–8] and references therein). Although the first tabulated results involving small numbers of cluster elements (n) were obtained well before the modern computer era [1], due to the very rapid increase of the number of configurations, the solution of this problem cannot be obtained without the computer help even for a not too large n. The classic enumeration algorithm proposed by Martin [3] allowed [9] to obtain all two-dimensional rectangular lattice cluster configurations up to $n = 19$. A *FORTRAN* implementation of Martin's algorithm, for the same kind of lattice, has been developed by Redner [4]. Using one of the two algorithms described in the present study, Shneidman and Nita [8] obtained the complete number of configurations and their distribution with regard of the number of the first neighboring bonds for $n \leqslant 17$. More recently, using a modified version of Redner's implementation [4], [2] provided equivalent tabulated data up to $n \leqslant 21$.

Although valuable for the thermodynamical description of clustering systems, especially with regard of the critical size definition [8], only the number of configurations and their bond distribution do not provide complete information about the clustering system dynamics. To obtain this kind of information, one should investigate all possible transitions between clusters of different sizes. This approach have been used by Shneidman and Nita [6,7] who, to study the dynamics of an Ising system, produced all possible transitions involving adding or loosing one cluster element up to $n = 9$ and the lowest energy nucleation path up to $n = 31$.

Given the fact that extending the limits currently achieved in this kind of simulations may improve the accuracy of the outcome results, and that the applicability of these results may not be restricted to the particular field where they have been already employed, this study presents the original algorithms used by Shneidman and Nita [6–8]. To obtain new results beyond the current limits, these two algorithms may be used in their current form on more powerful computer systems, or they may be used in the future as a starting point in developing more efficient algorithms. To help reaching this goal, beside the implementation of these algorithms, detailed efficiency and optimization issues are also discussed.

2. CLUSTER algorithm

2.1. Geometrical considerations

The first algorithm presented in this paper (CLUSTER) has been developed with the goal of generating all distinguishable n-element cluster configurations possible to be formed in a two-dimensional rectangular lattice. One of its basic ideas is that any distinguishable n-element cluster formed in an infinite rectangular lattice can be translated inside a predefined $n \times n$ finite square lattice such as each of the two axes of the orthogonal system associated with one of the square corners is in touch with at least one cluster element. Any of these translated n-element clusters can be further inscribed into smaller $x \times y$ rectangles, where x and y represent, respectively, the actual width and height of the n-element cluster. Among these rectangles, a particular role is played by the largest $L \times L$-square subscribing an n-element cluster, whose side length is given by

$$L = n - \mathrm{int}(n/2), \tag{1}$$

where int(n/2) states for the integer part of n/2.

The importance of this particular square is given by the fact that any rectangle subscribing an n-element cluster must have at least one of its sides less or equal to L, while the other one may range anyway from 1 to n. Thus, we may divide the n-element clusters in two classes: one containing the clusters subscribed by rectangles having both sides smaller or equal to L, and the other containing the clusters subscribed by rectangles with one side less or equal to L and the other strictly larger than L but less or equal to n. These two classes are different in terms of a rotation transformation because a 90° rotation in the lattice plane of any cluster belonging to the second class is always distinguishable from the original configuration, while all the n-element cluster configurations that may, but not necessarily, have a 90° rotational symmetry are included in the first class.

All the above considerations allow us to reduce the problem of generating all distinguishable n-element cluster configurations inside an $n \times n$ rectangular lattice, to the problem of generating all n-element cluster configurations inside an $L \times n$ rectangular lattice, with L given by Eq. (1). All missed configurations due to the reduced lattice size can be recovered at the time of generation by a 90° rotation of each cluster having one of its sides larger than L.

2.2. Numerical representation

The problem we should now challenge by developing the CLUSTER algorithm is to find all distinguishable n-element cluster configurations possible to be formed inside an $L \times n$ rectangular lattice. Mathematically, this implies finding the valid cluster configurations among the $C^n_{L\times n}$ potential combinations that fill n of the $L \times n$ empty sites of the rectangular lattice.

For a reasonable small number of elements, there is always a straightforward brute-force approach available. Indeed, any discernable lattice configuration can be mapped in a unique ($L \times n$)-bit integer, whose each bit associated with an occupied lattice site is set to 1. By direct inspection of all integers ranging from 1 to $N_{\max} = 2^{L\times n} - 1$, all discernable and valid n-element cluster configurations can be easily identified, although the computation time required to generate all $N_{\max}$ integers may become longer than the human life-time for fairly small values of n. That makes this brute-force approach inefficient for more ambitious results. However, the basic idea contained in the brute-force approach described above may be used as the starting point in developing the more efficient algorithm presented in this section.

An alternative to the ($L \times n$)-bit integer representation of the lattice configuration is the mapping of the same lattice into an n-element array with each element being an L-bit integer. In this representation, each

Table 1
The boolean table representation of a 5-element cluster

0	0	0	5
0	1	0	4
0	1	0	3
0	1	0	2
0	1	1	1
3	2	1	

such integer maps one of the n rows of the $L \times n$ boolean table associated with the rectangular lattice. As an example, Table 1 displays a 5-element cluster in a boolean table representation.

The boolean table represented in Table 1 has an horizontal index ranging from 1 to 3 = 5 − int(5/2), and a vertical index ranging from 1 to $n = 5$. For convenience, and compliance with the standard binary representation, the horizontal index runs from right to left, while the vertical one runs from bottom to top. The one dimensional representation of the same 5-element cluster is given by the 5-element array $c_5 = [3,2,2,2,0]$, where the array index ranges from 1 to $n = 5$ and each element may range from 0 to $2^3 - 1$. Since the 5-element cluster represented in Table 1 has a height larger than $L = 3$, it has an associated 90° rotated configuration, $\tilde{c}_5 = [15, 1, 0, 0, 0]$, which would have to be added to the set generated inside the reduced 3×5 rectangular lattice in order to assure the completeness of the 5-element cluster collection.

In conclusion, any n-element cluster generated inside a reduced rectangular lattice may be mapped into an n-element integer array, c_n, with an initial set of constraints given by

$$0 \leqslant c_n[j] \leqslant 2^L - 1, \quad 1 \leqslant j \leqslant n, \qquad L = n - \mathrm{int}(n/2). \tag{2}$$

2.3. Cluster validation tests

In order to represent a valid cluster configuration, any c_n array, as defined in Eq. (2), must fulfill a set of necessary conditions. A set of validation tests developed for this purpose are presented below.

The first validation test checks whether or not the total number of 1 bits mapped into the c_n configuration is equal to the required number of cluster elements, n. This test may be performed making use of basic bitwise operations

$$\sum_{j=1}^{n} \sum_{i=0}^{L-1} \{c_n[j] \text{ AND } 2^i\} = n. \tag{3}$$

The second validation test verifies whether or not the potential c_n cluster is distinguishable from other configurations in terms of a translation transformation. As discussed in Section 2.1, this condition requires at least one element to be in contact with each of the orthogonal axes of the coordinate system associated with the reduced rectangular lattice. This constraint is fulfilled if both of the following conditions are true:

$$\begin{aligned} &c_n[0] \geqslant 1, \\ &\textstyle\sum_{j=1}^{n} (c_n[j] \text{ AND } 1) \geqslant 1, \end{aligned} \tag{4}$$

where the first condition assures that there is at least one bit set in the bottom row of the boolean table, while the second condition assures that there is at least one bit set on the first column of the same table.

The third validation test checks if the c_n configuration is compact enough to represent a potential valid cluster. A minimum requirement is to have at least one connection between two adjacent vertical levels situated below the highest occupied one. This condition may be expressed as

$$c_n[k] \text{ AND } c_n[k-1] \geqslant 1, \quad 1 < k \leqslant j_{\max}, \quad c_n[j > j_{\max}] = 0, \tag{5}$$

which also implies that all nonzero elements of any valid c_n array always form a compact zone that occupy its first positions, while all the zero elements, if any, occupy the last positions of the array.

If all of the tests described by Eqs. (3)–(5) are passed by a given lattice configuration, it is worth trying to perform the last necessary test, which consists of checking if all n elements are linked together such us to form a valid cluster. The contiguity test is more demanding than the previous ones and its actual implementation may not have a unique solution.

The mechanism used in the present algorithm is to initiate a limited action broadcast signal, starting from one of the occupied sites and able to propagate only one lattice constant in all of the four orthogonal directions. Any occupied first order neighbor reached by the signal jumps from a not flagged in a flagged state and, recursively, sends a similar signal able to be received only by its not flagged yet first order neighbors. In this way, the broadcasting stops when all contiguous sites liked with the starting element are flagged, while any isolated element or group of elements remain not flagged. Counting the total number of sites reached by the broadcasted signal, the lattice configuration is validated as an n-element cluster if the number of flagged sites equals n. The same broadcasting routine may perform more than one useful task in the same time. It can be used not only to validate a cluster configuration, but also to return the number of bonds inside the cluster, the number of bonds of each particular element, or the actual width and height of the cluster. All information of this kind may be later used if demanded by the particular goal of the simulation.

2.4. Description of cluster algorithm

As shown before, the brute-force approach, i.e., the generation of all discernable lattice configurations and selection of valid clusters, cannot be performed for large cluster sizes in a reasonable amount of time. The more effective approach used in the present algorithm makes use of a recursive procedure that automatically cuts off any of its branches identified as not being able to give rise to at least one valid cluster configuration. The basic steps of this algorithm are listed below:

1. Setup the lattice limits (n; $L = n$ div 2 + n mod 2) and start from the bottom level ($j = 1$) of an empty input lattice ($c_n[j] = 0,\ j = \overline{1,n}; s_0 = 0$), where s_0 is the number of the already occupied sites in the input lattice.
2. Initiate a loop in which the column index, i, runs from 1 to $i_{\max}$, where $i_{\max} = 2^L - 1$ if $j = 1$, or it may have a different value if $j > 1$, as it will be explained later.
3. Generate a j-level configuration, $l_j = (i \text{ SHL } d_j)$, where the bitwise operator SHL generate the integer l_j by shifting d_j bits to the left the integer i. If $j = 1$, then $d_j = 0$, while it may have a different value when $j > 1$, as it will be explained later.
4. If $d_j > 0, j > 1$, and $i = 1$, check if $\sum_{k=1}^{j-1}(c_n[k] \text{ AND } 1) \geqslant 1$ to assure that at least one proper translated cluster may be generated at the current level. If the test is failed exit the loop and return to the previous level.
5. If $j > 1$, check if $(l_j \text{AND} c_n[j-1]) \geqslant 1$, to assure that the current level configuration may connect in a least one point with the lower level. If the answer is "NO", go to 14.
6. Generate a new lattice configuration, by setting $c_n[j] = l_j$, and compute the number of elements in the new configuration, $s = s_0 + \sum_{i=0}^{L-1}(l_j \text{ AND } 2^i)$.
7. If $s > n$, go to 14.
8. If $s < n$, answer the question "MAY c_n BECOME A VALID CLUSTER AT THE NEXT LEVEL?".
9. If the answer is "NO", go to 14.
10. If the answer is "YES", initiate a recursive call of the same procedure, entering it at the Step 2 with a level index equal to $j + 1$, an input lattice identic with the current c_n, and an input number of already set elements $s_0 = s$.
11. Since, at this point, $s = n$ and $c[1] \geqslant 1$, check if $\sum_{k=1}^{j}(c_n[k] \text{ AND } 1) \geqslant 1$ to assure proper translation. If the test is failed go to 14.
12. Since, at this point, $s = n$ and the translation test has been passed, initiate the broadcasting procedure to check the validity of the current n-element configuration.
13. If the cluster is valid, add it to the collection along with all necessary information about it. If the cluster height is larger than L, i.e., $c_n[L + 1] \geqslant 1$, generate the transposed cluster and add it to the same collection.
14. If $i \leqslant i_{\max}$, make $i = i + 1$ and go to 3, otherwise return to the previous level if $j > 1$, or STOP the program if $j = 1$.

2.5. Optimization issues

The net advantage of the algorithm presented in the previous section is that, compared with the brute-force approach, it is less time consuming and open for further optimization. The goal of this section is to discuss some of the optimization procedures featured by this algorithm, or possible to be implemented in more optimized versions.

Firstly, it can be observed that any new generated valid cluster configuration, as defined in the previous sections, is a new distinguishable cluster. This is assured by the fact that no duplicate lattice configuration may be generated by this algorithm.

Moreover, the recursive level approach allows for the most unproductive branches to be cut off from the beginning, because the algorithm disregard any lattice configuration found unable to lead to a valid cluster configuration. This optimization is implemented inside several key steps of this algorithm.

2.5.1. Loop initialization

The first optimization regards the loop initialization (Step 2), having the goal of reducing as much as possible the number of lattice configurations needed to be generated in order to produce a given valid cluster. Without any optimization, to produce any valid c_n cluster of height j, it would be necessary to initiate at least j nested loops, each of them running from 1 to $2^L - 1$. This would imply checking $(2^L - 1)^j$ lattice configurations in order to generate a collection of valid configurations containing the desired cluster. However, it can be shown that all but the first level loop may be in fact dynamically initialized over a much shorter range of index values, without loosing any valid configuration.

As an example, to obtain all the 16-element clusters of height 4 starting from the configuration shown in Table 2, it is not necessary to initiate a loop ranging from 1 to $2^L - 1 = 255$. Having only two more elements available to complete a 16-element cluster, they cannot be connected with the level 3 unless both of them lays, connected to each other, in the range marked by "?" symbols.

Thus, for this particular example, the 4-level loop has to be performed only $15 = 2^4 - 1$ times in order to generate all 15 configurations potentially connected with the previous level. From these 15 configurations, only three 2-bit configurations would complete the cluster at the level 4, while each of the only two 1-bit configurations connected with level 3 would initialize a 5-level loop to complete a 16-element cluster. However, each of these two 5-level loops would have to be performed only once, since the single element left in each case at the fifth level may be added only on top of the single element existing at the level 4.

Table 2
A 16-element cluster optimization example: setting the loop range

0	?	?	?	?	0	0	0	4
0	0	1	1	0	0	0	0	3
0	1	1	1	1	0	0	0	2
1	1	1	1	1	1	1	1	1
8	7	6	5	4	3	2	1	

The idea presented above has a straightforward generalization. Given a starting l_{j-1} configuration, the column indexes of the first, $b_{\min}$, and the last, $b_{\max}$, occupied sites of the l_{j-1} configuration are computed. Using the maximum number of bits available at the next level, $a_j = N - s_0$, the bit indexes of the valid range are computed as:

$$\begin{aligned} r_{\min} &= \max(b_{\min} - a_j + 1, 1), \\ r_{\max} &= \min(b_{\max} + a_j - 1, L). \end{aligned} \tag{6}$$

In order to generate all configurations inside the valid range, a j-level loop of index i, running from 1 to $i_{\max} = 2^{r_{\max} - r_{\min} + 1} - 1$, is initiated, and a shift constant, $d_j = r_{\min} - 1$, is defined. For each loop index i, a potentially connected j-level configuration is generated as

$$l_j = i \,\mathrm{SHL}\, d_j. \tag{7}$$

Although, in some cases, the computation time spent at this stage may not result in a change of the loop dimension at all, from a global point of view, the total running time of the algorithm may be significantly reduced, since any single configuration skipped at a j-level may eliminate as much of $(2^L - 1)^{(a_j - 1)}$ nonproductive lattice configurations, which would be otherwise generated without no gain.

2.5.2. Optimization checkpoints

Beside the initial loop optimization, the algorithm contains three more checkpoints that efficiently reduce the computational time. The first checkpoint is located at Step 4 and it is performed only if the shift constant is not zero ($d_j > 0$), the current level is not the ground level ($j > 1$), and the current level loop has been just initialized ($i = 1$). In such a particular situation, it is worth to check if it may exist at least one level configuration able to produce a j-level potential cluster properly translated such as to connect with the vertical axis. The importance of such test resides in its powerful potential outcome: had the test failed, an entire nonproductive j-level loop, just initialed to range from 1 to $i_{\max}$, and all higher level attempts based on its configurations, as well unproductive, would be avoided. Table 3 exemplifies such a situation in the case of a 16-element cluster generation attempt.

Starting with the 3-level configuration given in Table 3, an $i = \overline{1, 127}$ loop is initialized at the fourth level. However, since none of the previous levels is connected with the vertical axis, none of the 127 configura-

Table 3
A 16-element cluster optimization example: checking the translation constraint

?	?	?	?	?	?	?	0	4
0	0	1	0	1	0	0	0	3
0	1	1	1	1	0	0	0	2
1	1	1	1	1	1	1	0	1
8	7	6	5	4	3	2	1	

tions produced by this loop can produce a properly translated cluster. Moreover, any of these 127 configurations having less than 3 elements would initialize a 5-level loop attempting to complete a valid 16-element cluster configuration. The process would be continued up to the sixth level, with no chance of producing a properly translated 16-element cluster. Generally, we may conclude that if any of the most right translated configurations at a given level, in this example $c_{16}[4] = 14$, cannot be connected with the vertical axis, all of the other configurations at the same level, or possible to be produced at a higher level, cannot be connected with the vertical axis either and, therefore, the current level can be exited without risking to miss any valid cluster configuration. This test is done by checking the condition

$$\sum_{k=1}^{j} c_n[k] \geqslant 1 \tag{8}$$

and returning to the previous level without performing the loop, if the test is failed.

The second checkpoint is located in the Step 5 where, unless on the ground level, the loop horizontal index is incremented if there is no physical connection with the previous level, as required by a potential cluster configuration. The function performed by this test is similar with the one performed by the loop initialization, with the only difference being that, instead of discarding an entire block of unproductive level configurations, each failed test discards, one by one, all the unproductive configurations that lay inside the valid loop range.

The third checkpoint is located in the Step 8, where a new level is initialized only if the current configuration, already connected in at least one point with the previous level, may really become a valid cluster at the next level. The answer at this question may be obtained in a more or less sophisticated manner, depending on the optimization degree desired. The minimal test, used in this implementation as a necessary but not sufficient condition, therefore still open to improvement, checks if there is at least one potential combination available at the next level that would be able to connect all already existent elements, even those not connected with the previous level, in a single cluster of any size. Evidently, the simplest connecting bridge one can imagine, is a compact line configuration, $2^L - 1$, placed on top of the existing level. The new potential configuration, $\tilde{c}_n$, is used to initiate a broadcasting procedure that checks if the condition

$$\sum_{k=1}^{j+1} \sum_{i=0}^{L-1} \{\tilde{c}_n[k] \text{ AND } 2^i\} = s + L \tag{9}$$

is fulfilled. The failure of this test automatically implies a negative answer to the question "MAY c_n BECOME A VALID CLUSTER AT THE NEXT LEVEL?" and the next level is not initialized. However, if the test is passed, there is no guarantee that the number of elements available at the next level may assure the existence of at least one bridge configuration able to connect the existing lattice state in a cluster of desired size. Table 4 displays an example of such unproductive situation unable to be detected by the minimal tests described above.

The initial configuration presented in Table 4, intended to produce 16-element clusters, generates three nested loops, all running from 1 to 255, and checks 765 configurations up to level 6, to finally reach no valid 16-cluster configuration. These unproductive steps could be avoided if the first attempt to reach the superior level would fail just because, with only 6 more elements left, no valid 16-element cluster configuration can be generated, even if all of the available elements would be employed at the fourth level. However, the 765 unproductive steps performed in this example are several order of magnitude fewer than the number of unproductive configurations that would have been generated in the absence of the minimal implementation of the "MAY CONNECT?" and "MAY BECOME?" checkpoints. On the other hand, one should always have in mind the possibility of obtaining an undesired result, i.e., slowing down the algorithm, by implementing more sophisticated checkpoints that, although may solve particular situations such as that

Table 4
A 16-element cluster optimization example: checking the potentiality to become a valid cluster

0	?	0	0	0	0	?	0	6
0	?	0	0	0	0	?	0	5
0	?	0	0	0	0	?	0	4
0	1	0	0	0	0	1	0	3
0	1	1	0	0	0	1	0	2
0	1	1	1	1	1	0	0	1
8	7	6	5	4	3	2	1	

described in Table 4, may require a computational time competing with the time that would be required to generate all the unproductive branches intended to be discarded.

2.5.3. CLUSTER algorithm efficiency

The efficiency of this algorithm may be evaluated comparing the number of lattice configurations needed to be generated in order to obtain all distinguishable valid cluster configurations. As an example, in order to obtain all 505,861 distinguishable 12-element cluster configurations, the algorithm generates 17,248,733 lattice configurations that are checked for validity by one or all of the checkpoints presented above. Using these values, a global algorithm efficiency of about 2.93% may be computed. To fully appreciate it, this result has to be compared with the efficiency that would have been obtained in the case of a brute-force approach generating all distinguishable lattice configurations, which are in this particular case as many as $2^{6\times 12}$.

A more detailed analysis is presented in Fig. 1, where the generated and valid number of configurations are compared for each first level iteration index, which ranges from 1 to $63 = (2^6 - 1)$. The (a) panel presents the normalized distribution of the generated (thin line) and valid (thick line) configurations, versus the first level loop index. The normalization factor has been chosen to be the total number of generated lattice configurations (trials). A good correlation between the compared magnitudes may be observed, although the distributions are not uniform. The (b) panel plots the algorithm efficiency at each iteration (thick line), which is compared with the median efficiency (thin line), given by the value above or below 50% of the individual ratios are distributed. The systematic change in this distribution, which lays above the median value for the first half of loop indexes, while it lays below the median value for the second half, is manly due to the fact that only the configurations produced in the first half of the first level loop may have a height larger than $L = 6$, in which case they automatically generate their 90° rotated configurations without requiring supplemental computations. The local maxima or minima that appear in the distributions presented in both

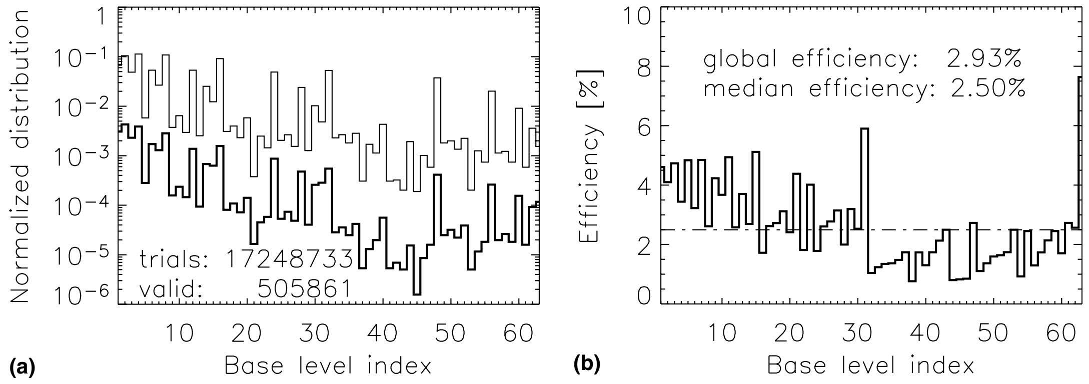

Fig. 1. CLUSTER algorithm performance in the case of a 12-element cluster generation. (a) Normalized distributions of the number of generated lattice configurations (thin line) and the number of valid cluster found (thick line), versus the iteration index of the algorithm first level loop. The total number of lattice configurations generated by the algorithm is used as normalization factor. The normalization factor (trials) and the total number of valid cluster configurations found (valid) are indicated in the figure. (b) Algorithm efficiency. The valid to trials ratio (thick line) and the median efficiency (dashed-dotted line) are plotted versus the first level loop index. The median and global efficiency are indicated on the figure.

panels are dictated by the efficiency of the various constraints imposed by the algorithm checkpoints, which depends on the number of first level bits, or on the maximum distance between them.

2.5.4. Parallel processing

Although the minimal optimization procedures presented above may identify most of the already dead branches of the recursive algorithm at the earliest stage possible, helping to reduce the computation process by a considerable amount of time, they may not be enough to allow obtaining the desired results for a not too small number of cluster elements n. As an example, the version of this algorithm strictly based on the optimization methods presented above, written in C++ and run on an up to date desktop computer, required a computational time ranging from less than a second, for $n \leqslant 10$, up to about 18 days, for $n = 16$.

However, the computational time performance of CLUSTER algorithm remains significantly modest compared with the classical enumeration algorithm developed by Martin [3]. Indeed, using a C++ implementation of Redner's code [4], modified such as to remove the degeneracy related to the number of bonds, it was possible to obtain, on the same computer system, the full 16-element cluster collection in only 87 s.

Although the above comparison suggests that the CLUSTER algorithm cannot compete with the most efficient enumeration scheme of Martin [3], one should be aware that, for not too much larger cluster sizes, any sequential approach would not be able to overcome the physical time required to produce once at the time a new cluster configuration, even with 100% efficiency.

Since for some particular applications, being able to increase the cluster size even by one element may make a big difference in terms of the potential outcome, it is worth trying to find an alternative to the sequential enumeration scheme developed by Martin [3].

Fortunately, the CLUSTER algorithm, even in the form presented above, is intrinsically suitable for a parallel processing option. Indeed, the system of nested loops initialized by this algorithm can be split into totally separated threads that can be run on separate computers, or microprocessors. The parallel processing approach is allowed by the fact that each individual first-level configuration produced by the algorithm's main loop initializes a recursive call completely separate from the others, which finally returns to the same first level. Therefore, instead of waiting for each thread to be finished in order to generate the next first-level configuration, each of these first-level configurations can be used to launch separate parallel processes, whose results can be simply added after all threads are performed. Moreover, any higher level configuration can be used as the root of a new parallel process, therefore, the algorithm can be, at least in principle, split in supplementary parallel processes at any configuration level.

In conclusion, the CLUSTER algorithm, when used to generate all distinguishable n-element cluster configurations, can be split in at least $2^{n - \mathrm{int}(n/2)} - 1$ first-level parallel processes, whose partial results may be finally combined to get the complete cluster collection. However, one should be aware that the computational time required to run all of the parallel processes would not decrease proportionally with the number of independent threads, because each particular first level configuration may require a totally different computational time. Therefore, the gain of using parallel processing is dictated by the longest independent process that is initialized. As an example, the distribution of algorithm trials presented in Fig. 1(a) (thin line) may be also regarded as the distribution of the fractional computational time needed by each independent thread. A much uniform thread time distribution may be obtained by dividing the first level loop in groups of smaller loops which may be used to launch a reasonable number of separate threads.

Although a true parallel process has not been yet implemented for the CLUSTER algorithm, by manually launching independent threads on separate computers, it was possible to obtain the full 17-element cluster collection in a much shorter amount of time than the 18 days needed to obtain, without parallel processing, all of the 16-element cluster configurations.

While still a modest time performance, if compared with Martin's algorithm [3], this result at least shows that the parallel implementation of the CLUSTER algorithm may be a viable solution able to succeed beyond the natural limits of any sequential approach. The future efforts in this direction are fully motivated since, for some particular scientific applications, being able to increase the cluster size even by one element may make a big difference in terms of the potential outcome.

2.6. CLUSTER algorithm results

The algorithm presented in this section has been successfully used to produce all discernable cluster configurations up to 17 elements. This limitation has been entirely dictated by the computational time required to produce them. However, as explained in the previous section, nor the possibility of obtaining more ambitious results, neither the possibility to obtain them in a much shorter amount of time, are excluded.

Table 5 displays the exact number of discernable configurations for cluster sizes ranging from 1 to 17 elements. As expected, the order of magnitude of these numbers is rapidly increasing with the cluster size. Table 6 displays the distribution of the number of clusters of any size with a given number of bonds. Since the possible number of bonds for a given cluster size n may start as low as $n - 1$, the distribution presented in Table 6 is exact only for bond numbers ranging from 2 to 16, inclusive. The incomplete results due to the not generated yet configurations above 17 elements are indicated by asterisks.

Since the challenging mathematical problem of finding the expressions able to generate all these results and the subsequent ones may or may not have an exact solution, we can at least check if these numerical series are suitable for numerical estimations of the subsequent values. Fig. 2 reveals that both kinds of distributions are well approximated by exponential–quadratic functions with comparable exponential coefficients. Although relatively small, the quadratic coefficients are essential for an accurate estimation [8]. Therefore, using the present results, one may estimate at least the order of magnitude of the results involving larger cluster sizes than currently reported. As an example, the estimated number of distinguishable 18-element cluster configurations inferred from this dataset is 1,715,247,360 ± 508,225,792, while the estimated number of clusters having 17 bonds is 774,388,800 ± 176,322,512. These estimated values are indicated on each panel of Fig. 2 (error bars). However, these values should be used only to estimate the amount of computer resources and computational time that would be required to generate collections of larger cluster sizes than those presented in this study.

Table 5
Cluster algorithm results: number distribution

Number of elements	Number of clusters
1	1
2	2
3	6
4	19
5	63
6	216
7	760
8	2725
9	9910
10	36,446
11	135,268
12	505,861
13	1,903,890
14	7,204,874
15	27,394,666
16	104,592,936
17	400,795,840

Table 6
CLUSTER algorithm results: bond distribution

Number of bonds	Number of clusters
1	2
2	6
3	18
4	56
5	182
6	610
7	2078
8	7172
9	25,044
10	88,272
11	313,444
12	1,120,084
13	4,024,724
14	14,530,672
15	52,678,550
16	191,678,276
17*	158,894,206
18*	80,919,452
19*	28,541,146
20*	7,788,054
21*	1,650,792
22*	281,218
23*	36,330
24*	3011
25*	88

The asterisks denote incomplete results as explained in the text.

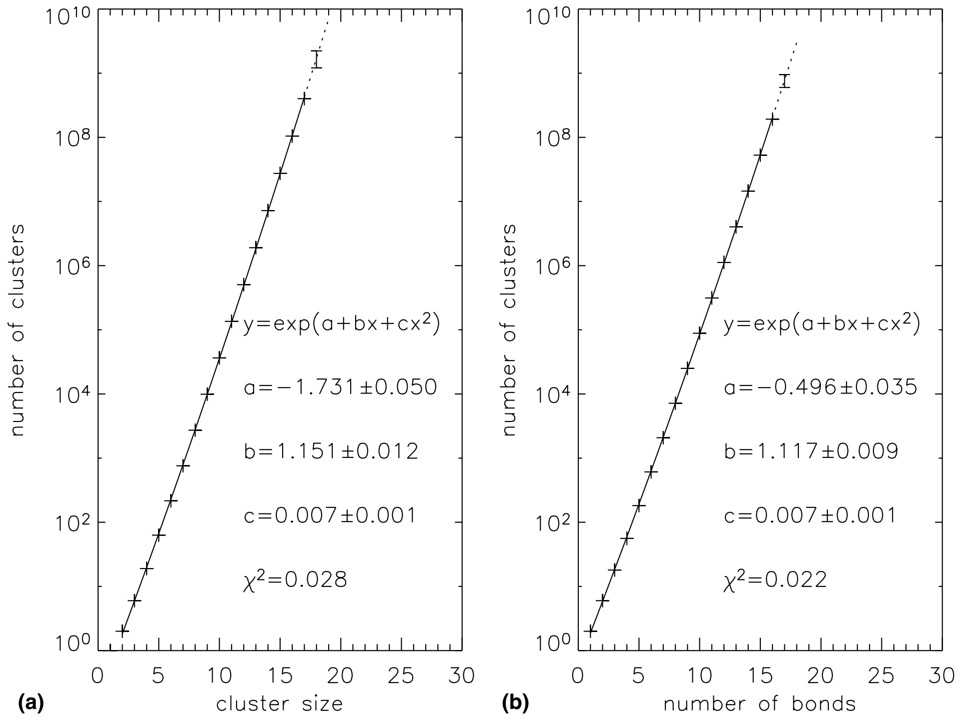

Fig. 2. CLUSTER algorithm results: (a) The number of discernable cluster configurations (ordinate) versus the number of cluster elements (abscissa), as presented in Table 5. (b) The number of clusters (ordinate) with a given number of bonds (abscissa), as presented in Table 6. The data points corresponding to more than 16-bonds, which are incomplete results as explained in the text, are not indicated on (b) panel. Both distributions are well fitted by exponential–quadratic functions with comparable coefficients (the $1 - \sigma$ standard deviations of these coefficients and the goodness-of-fit χ^2 are indicated on each panel). The extrapolated functions are shown on each panel (dotted lines), while the estimated values corresponding to the not computed yet 18-element cluster size, and their standard errors, are shown on each panel (error bars).

3. GROW algorithm

As mentioned in Section 1, the results provided by any enumeration approach, including the above discussed CLUSTER algorithm, may be useful in thermodynamical applications involving the distribution of n-element cluster properties (see [8] as a study based on the results presented in the previous section). However, such algorithms are not able to provide, at least in the form presented in this study, any information involving the possible transitions between two clusters of different sizes. Such kind of information, which may be of special interest in any application involving the dynamics of cluster systems [6,7,5], requires a different approach that has been implemented in the alternative algorithm presented in this section.

As suggested by its name, the GROW algorithm is able to generate, at least in principle, all the cluster configurations that can be formed by adding, one by one, a given number of elements to an already existing configuration, referred from now on as a nucleus. Evidently, the simplest possible nucleus is an isolated element that, in the particular case of a rectangular lattice, can accept as many as four other elements to be directly attached to its unsatisfied bonds. Subsequently, each of the attached new element may accept new elements to be attached to its empty neighboring sites, and the process may be continued until a preset maximum cluster size is reached.

Although the basic idea of this algorithm is extremely simple, its actual implementation is more computationally demanding than that of the CLUSTER algorithm. This is because, unlike in the previous case, the

GROW algorithm may produce duplicate configurations and, therefore, it requires the comparison of any new generated configuration with all the configurations already produced, which has to be stored in a continuously expanding database. It should be mentioned at this point that, although the *growing* scheme of this algorithm is in many aspects similar with the sequential approach of Martin [3], it essentially differs from the latest by the fact that its purposely generated duplicate configurations are used to identify all possible transitions between two given clusters of different sizes. For this reason, it becomes evident that GROW algorithm's performances may be limited not only by the computational time, but also by the physical memory available to store, in a distinguishable representation, all of the already generated configurations. However, as it will be explained shortly, this algorithm is more flexible than the previous ones, allowing the user to obtain useful partial results for much larger cluster sizes, if a set of selection rules is properly defined.

3.1. Definition of a cluster class

As mentioned above, the implementation of GROW algorithm requires a continuously expanding database, in which each distinguishable configuration newly produced has to be added in a unique form containing all useful information about it, which we will refer as a cluster class.

Since the performance of this algorithm may be influenced by the physical size of the cluster class database, it is worth trying to reduce as much as possible the number of classes that has to be stored, which can be done by allowing more than one distinguishable configuration to belong to the same class. However, depending on the particular goal of the simulation, the algorithm remains open to a redefinition of a cluster class in the sense of increasing or decreasing its degree of degeneration.

In this study, the same cluster class definition as in [6,7] is employed. According with this definition, all cluster configurations which can be obtained from each other by using a geometrical symmetry transformation are allowed to belong to the same class. Therefore, the degeneration degree of such a class cannot exceed 8, which corresponds to a maximum of four 90° rotations multiplied by two reflections that can be employed in a two-dimensional rectangular lattice. The minimum degeneration degree of a such class is 1, which corresponds to a perfect square configuration, while a perfect rectangular configuration would have a degeneration degree equal to 2.

Based on this cluster class definition, each database record requires a minimum list of characteristic parameters, which are listed below:

1. An ID number, used to uniquely identify the cluster class.
2. The number of elements, **n**.
3. The number of neighboring bonds, **b**.
4. The degeneration degree, or geometrical weight, **w**,
5. The binary representation of the firstly produced geometrical configuration, which, for a rectangular lattice, can be in a form of a c_n array, as described in Section 2.2.
6. A From list of the $(n-1)$-element classes' IDs from which the elements of the current class can be grown by adding one element to one of their **w** distinguishable configurations.

3.2. Geometrical transformation routine

Beside the class description, the algorithm requires the definition of a geometrical transformation routine to be used for obtaining all potentially distinguishable geometrical configurations based on the binary representation stored in the class description. This transformation routine can be a stand alone one, or it can be encapsulated in the class definition, if an object-oriented approach is used.

In the case of a rectangular lattice, such a routine should produce 8 geometrical configurations by employing rotation and/or reflection transformations. However, one should notice that the basic idea of this algorithm is not restricted to the rectangular lattice case. Indeed, by redefining the transformation rules according to the actual lattice unit vectors, any two dimensional lattice type can be inspected. In fact, the present algorithm was successfully employed to generate cluster configurations in triangular and hexagonal two dimensional lattices, not discussed here. Moreover, the same algorithm may be, at least in principle, easily adjusted to generate three dimensional cluster configurations, although the number of possible configurations, even in the case of a modest number of cluster elements, may be too challenging for an usual computer system.

Whichever the lattice geometry, the geometrical transformations should be performed in such a way as to permit an easy comparison of different configurations. For this reason, any rotation or reflection should be accompanied by an appropriate translation, as described in Section 2.1.

3.3. Description of the GROW algorithm

The GROW algorithm is initialized by defining the starting nucleus on which the larger classes are grown. The simplest nucleus is a single isolated element, although any more complex structure can be chosen to start with. Once the nucleus is chosen, the class associated with its configuration is also defined and recorded in the database. After the initialization step, the functionality of the algorithm is based on a core procedure which is recursively called in order to generate all desired cluster classes possible to be grown starting with the given nucleus. The basic steps of this procedure are listed below:

1. For each element of the current nucleus, the neighboring sites are inspected to find the empty locations in which a new element can be added. For an n_0-element nucleus in a rectangular lattice, this step requires to initialize a loop whose index runs from 1 to n_0, with each iteration inspecting the 4 potentially empty neighboring sites of the currently investigated cluster element.
2. If such an empty site is found, a new n-element configuration is produced ($n = n_0 + 1$) by adding a new element to the current configuration, otherwise the next neighboring site is investigated.
3. Having produced a new **n**-element configuration, with **b** neighboring bonds, the transformation procedure is called in order to produce the **w** distinguishable configurations associated with its class.
4. The database is searched for all recorded classes to find those having the same **n**, **b** and **w** parameters.
5. For each record of this type found, if any, its binary representation is superposed on each of the **w** distinguishable configurations produced by the transformation routine, in order to check if the current configuration's class is already recorded in the database.
6. If no matching class is found, a new class is defined and added to the database. The From list of the new class is initialized with the ID number of the generating class. If the current configuration size is less than the maximum cluster size desired, a next level call of the same procedure is initialized with the current n-cluster configuration used as a starting nucleus.
7. If a matching class is found, its From list is searched and updated if the ID number of the n_0-element generating class is not already in the list. Since reaching this step implies that the current n-element configuration has been already produced at least once, thus already used as a nucleus to initialize an upper level growing process, and no new configuration can be grown on it, the search for a new empty neighboring site continues at the same level.
8. If all the empty neighboring sites of the current nucleus have been investigated, the procedure returns to the previous level, or the search ends, if this is its first level call.

3.4. Efficiency and flexibility issues

Unlike the CLUSTER algorithm, which must be run separately for each cluster size, the GROW algorithm generates all discernable configurations starting from the initial nucleus, up to the maximum size desired. Moreover, since each new element is directly linked to the initial nucleus, no broadcasting procedure is required to check the contiguity of the new generated configuration. However, the memory management and the search of the continuously increasing database may result in a much longer computational time required by GROW compared to the time required to generate the same number of configurations using the CLUSTER algorithm.

As an example, to generate all 13,702 distinguishable cluster configurations from 1 to a 9 elements, the total computational time required by CLUSTER was less than 2 s, while, on the same computer system, the GROW algorithm produced them in about 150 s. To better understand the influence of the database size on the algorithm's performance, it should mentioned that, while CLUSTER algorithm needed only 3 more

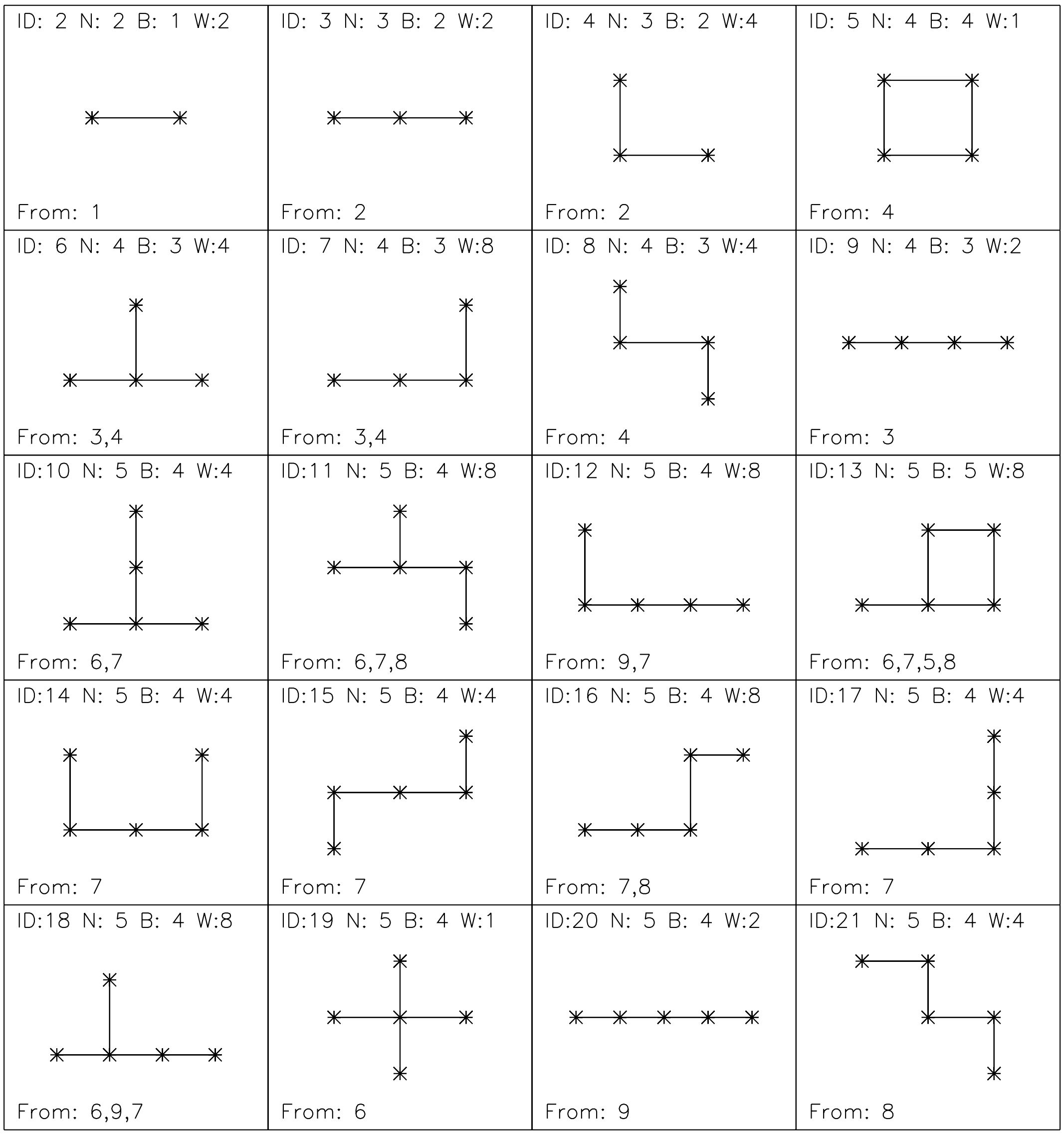

Fig. 3. GROW algorithm results: complete results up to 5-element clusters.

seconds to generate all of the 10-element configurations, the GROW algorithm needed about 2012 s to generate the full nucleation chains up to the same number of elements.

However, the much longer time required by the GROW algorithm may be compensated by valuable dynamical information about the investigated system, given by the classification of the cluster shapes (1818 classes up to 9 elements) and the finding of all possible transitions between them [6].

A net advantage of the GROW algorithm is given by its flexibility. Not only the growing process can start from any given nucleus, but also the nucleation branches can be considerably reduced if user defined selection rules, motivated by the underlaying physics of the process being investigated, are introduced in the algorithm.

As an example, a selection rule based on energetic considerations may be introduced by forbidding the formation of a new cluster configuration having a number of bonds too far from the maximum possible one, which corresponds to the most compact configuration possible to be obtained with *n* cluster elements. This selection rule may be easily implemented in the Step 2 of the GROW algorithm if no new element is added in an empty neighboring site given the number of bonds of the new configuration is less than the

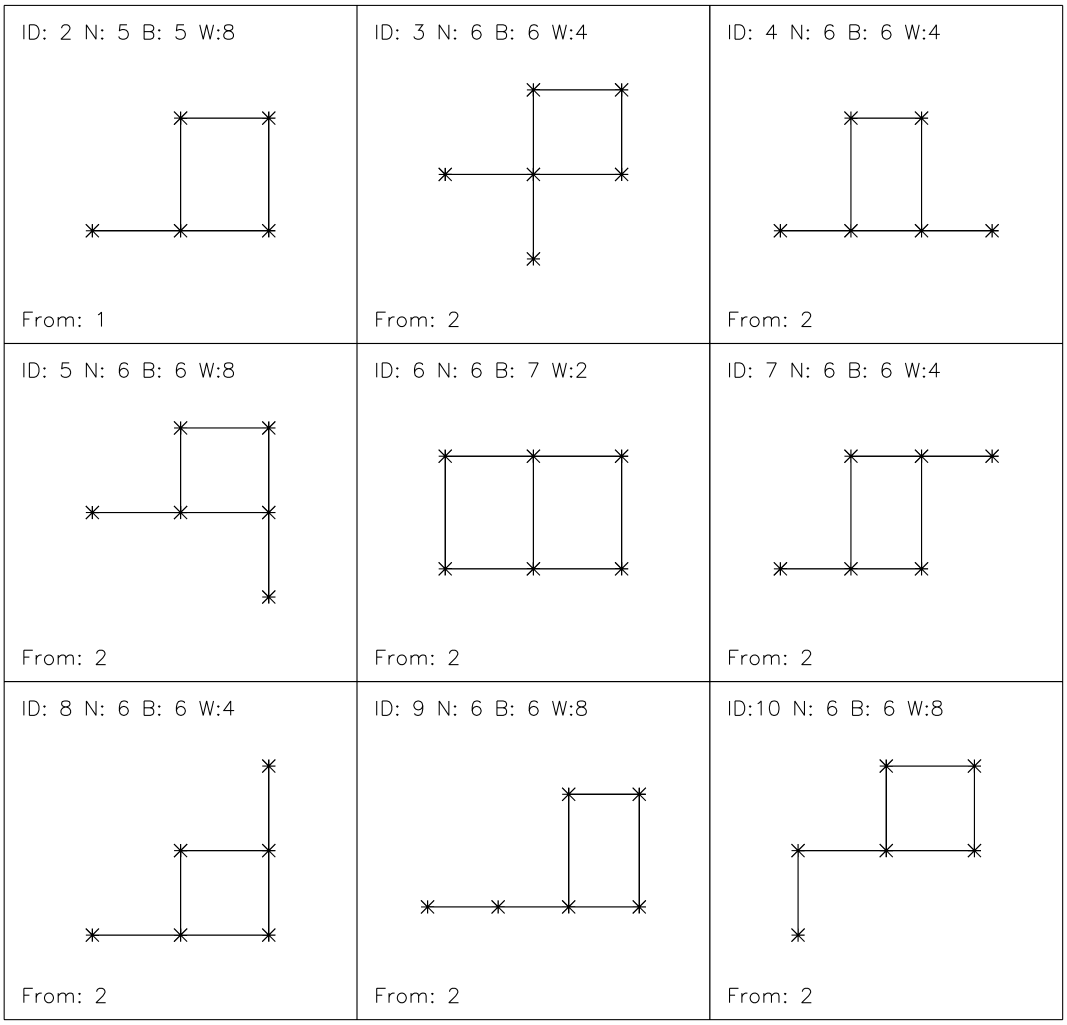

Fig. 4. GROW algorithm results: all 5 and 6-element classes connected with a 4-element perfect square nucleus.

difference between the maximum possible number of bonds and a preset reduction number Γ. Using this approach, Shneidman and Nita [7] were able to investigate the low-energy nucleation path in a dynamical Ising model up to 31 elements for $\Gamma = 0$, 19 elements for $\Gamma = 1$, and 12 elements for $\Gamma = 2$.

3.5. GROW algorithm results

As shown in the previous paragraph, the GROW algorithm has been successfully used in two particular applications [6,7] to generate complete results up to 9 elements and partial results up to 31. However, these limits should not be regarded as the true limits of the algorithm itself. In fact, the GROW algorithm

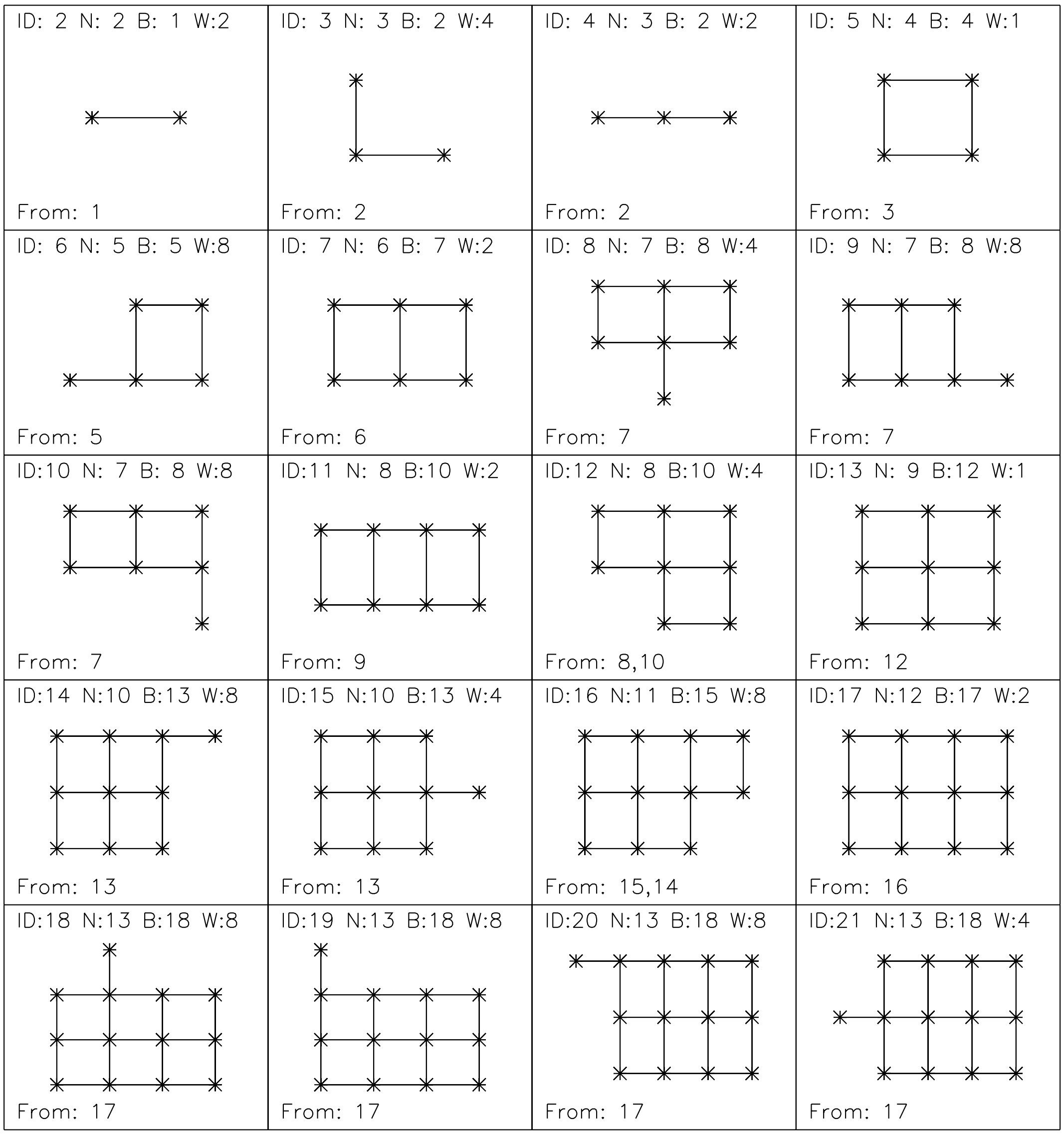

Fig. 5. GROW algorithm results: the most compact configuration chains up to 13 elements. Since some classes may be obtained from more than one nucleus, the bifurcation of the nucleation path may be observed even in this lowest-energy example.

successfully generated all 50,148 distinguishable configurations (6473 classes) up to 10 elements and the lowest energy nucleation path ($\Gamma = 0$) up to 100 elements (2098 configurations, 315 classes). However, for illustration purposes, due to the high number of configurations involved, only results obtained for much smaller cluster sizes are presented here in more detail.

Fig. 3 shows the full results up to 5 elements. Each panel shows one of the 20 classes that can be grown starting from an 1-element nucleus, which defines the ID = 1 class. In each panel, on the upper text line, the ID, the number of elements N, the number of bonds B and the geometrical weight W are indicated, while on the bottom text line the ID list of the classes from each the current one can be grown is displayed.

Fig. 4 displays all 5 and 6-element classes that can be grown starting with a 4-element perfect square nucleus. The information in each panel is the same as the one shown in Fig. 3.

Fig. 5 presents the lowest-energy chain from 1 to 13 elements. To obtain these classes, only the configurations with the maximum number of bonds were allowed to be formed for each cluster size [7]. One may notice that, even in this lowest-energy case, for some cluster sizes, there are not only more than one class with the same number of elements and the same maximum number of bonds, but also some classes may be obtained through more than one nucleation path.

4. Conclusion

In this study, two original cluster generating algorithms have been presented.

The CLUSTER algorithm uses a binary approach to produce all distinguishable n-element cluster configurations and their number of bond distributions. The results produced by this algorithm may be useful for thermodynamical applications, such as the one investigated by Shneidman and Nita [8]. The complete results up to 17 elements were presented. The current cluster size limitation depends only on the computational time required by the system used, but the possibility to obtain much ambitious results, using a parallel processing approach, were discussed.

The GROW algorithm uses a dynamical approach to generate the nucleation chains up to a preset maximum number of elements. In addition to the information produced by a simple enumeration scheme, this algorithm provides all possible transitions between the generated cluster classes. This kind of information may be of particular interest in dynamical applications, such us those presented in Shneidman and Nita [6,7]. It has been shown that, despite the GROW algorithm is more computationally demanding and its full results are limited by a smaller cluster size than the limit of any enumeration algorithm, valuable partial results, based on a set of user-defined selection rules, may be obtained for much larger cluster sizes. The possibility to generalize this algorithm to any kind of lattice geometry was also discussed.

References

[1] C. Domb, On the theory of cooperative phenomena in crystals, Adv. Phys. 9 (140) (1960).

[2] S. Frank, D.E. Roberts, P.A. Rikvold, Effects of lateral diffusion on morphology and dynamics of a microscopic lattice-gas model of pulsed electrodeposition, J. Chem. Phys. (in press). Available from: <cond-mat/0409518>.

[3] J.L. Martin, in: C. Domb, M.S. Green (Eds.), Phase Transitions and Critical Phenomena, vol. 3, Academic, New York, 1974, p. 97, 29P.

[4] S. Redner, A FORTRAN Program for cluster enumeration, J. Stat. Phys. 29 (1982) 309.

[5] V.A. Shneidman, On the lowest energy nucleation path in a supersaturated lattice gas, J. Stat. Phys. 112 (1/2) (2003).

[6] V.A. Shneidman, G.M. Nita, Modulation of the nucleation rate preexponential in a low-temperature Ising system, Phys. Rev. Lett. 89 (025701) (2002).

[7] V.A. Shneidman, G.M. Nita, Nucleation preexponential in dynamic Ising models at moderately strong fields, Phys. Rev. E 68 (021605) (2003).

[8] V.A. Shneidman, G.M. Nita, On the critical cluster in the 2-dimensional Ising model: computer-assisted exact results, J. Chem. Phys. 121 (11232) (2004).

[9] M.F. Sykes, M. Glen, Percolation processes in two dimensions. I. Low-density series expansions, J. Phys. A 9 (1976) 87.