J Math Chem (2009) 45:386–405
DOI 10.1007/s10910-008-9413-4

ORIGINAL PAPER

Optimized algorithm to find all symmetry-distinct maps of a graph: application to topology-driven molecular design

Erwin Lijnen · Arnout Ceulemans

Published online: 1 August 2008

Abstract We describe an optimized algorithm for finding all symmetry-distinct maps of a given graph. It contains significant improvements on the computing time by representing the maps as linear codes. In this way, the time consuming step of removing equivalent maps can be solved more efficiently by searching for a "minimal code". As an example we apply the algorithm to the 32-vertex Dyck-graph for which more than 4 billion cases should be investigated. One of its most symmetrical maps forms an interesting blueprint for a hypothetical negatively curved carbon allotrope of genus 3.

Keywords Topology · Graph-embeddings · Carbon · allotropes · Molecular design

1 Introduction

Since the early years of chemical theory, scientists have been familiar with mathematical graphs. In the first molecular models, a molecule was nothing else than a collection of solid spheres held together by attractive forces. This simplified model led to a very attractive pictorial representation of molecules as graphs [1], namely as a set of points (atoms) connected by lines (chemical bonds). Although this very crude approach leads to important insights into the nature of molecular structures, it completely neglects their 3D structure which is indispensable for the correct description of molecular properties like i.e. chemical activity. The most exact description of a molecule would be to give the exact positions of the nuclei together with a spatial distribution of the electrons. The storage of such exact coordinates is however extremely tedious if one is

E. Lijnen (✉) · A. Ceulemans
Departement Chemie and INPAC: Institute of Nanoscale Physics and Chemistry,
K.U. Leuven, Celestijnenlaan 200F, 3001 Leuven, Belgium
e-mail: erwin.lijnen@chem.kuleuven.be

working with large set of molecules like in the screening of large molecular databases for new lead compounds. This indicates that there is a clear need for an intermediate model which is not only combinatorial in nature, like the graph, but also incorporates some kind of metric. The most elementary way to add such geometrical information is by mapping the graph on a 2D closed orientable surface like the sphere or torus to form a *2-cell embedding* or *map* [2,3]. When such an embedding takes place, the graph divides the underlying surface in a set of 2D closed regions, called 2-cells or faces, which form a new entity not apparent in the purely graph theoretical description. Chemists have already become acquainted with the notion of a face through the structural description of inorganic complexes and especially through the flourishing field of carbon chemistry investigating structures like fullerenes and carbon nanotubes. The mathematical theory underlying this "polyhedral model" [4] is well-developed and known as the theory of oriented 2-cell embeddings or maps.

In a previous publication [5] we discussed how the polyhedral model can be used for the topology-driven design of novel molecular structures. In this process one first starts by searching for all symmetry-distinct maps of an interesting (in most instances a highly symmetrical) graph. In general, this set of maps will be very large and consists of maps on surfaces of different genera. Once this set is fully determined, one can isolate the most interesting maps (mostly the low genus or high symmetrical ones) and obtain their most symmetrical realization in 3D space. The procedure for finding such a 3D-realization was thoroughly discussed in Ref. [5]. In the present article we are mainly focused on the first step of finding all symmetry-distinct maps for a given graph. As the number of such maps grows exponentially with the size of the underlying graph we will need very efficient representations and algorithms to be able to investigate reasonably large graphs.

2 From map to rotation scheme

It is well known that a graph can be completely determined by its adjacency list, which for every vertex gives a complete list of all the vertices which are connected to it. A map[1] can be described in a similar way by means of a rotation scheme [3], which as we shall see, corresponds with an ordered adjacency list. The first step in forming a rotation scheme out of a given map consists in giving each point on the orientable surface (sphere, torus or higher genus surface) a consistent sense or handedness (clockwise or anticlockwise). An orientable surface provided with such a sense will be called an *oriented surface* and mappings on them are called *oriented maps*. Looking at the neighborhood of a vertex v on an oriented surface, the chosen sense determines an ordered sequence of edges around v, which is called a *local rotation* at v. This sequence is defined up to cyclic permutations. As a result, the number of possible local rotations at v, $n_r(v)$, is given by:

$$n_r(v) = (deg(v) - 1)! \tag{1}$$

[1] In the present paper we restrict ourselves to maps on orientable surfaces.

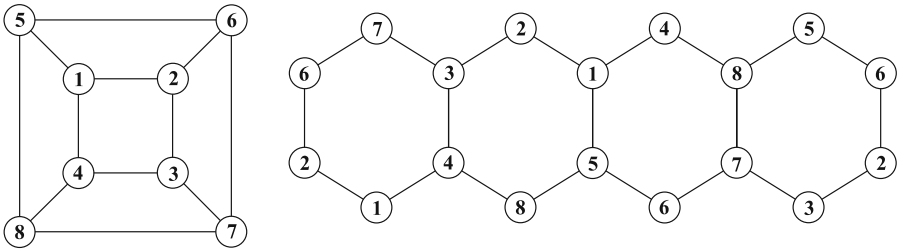

Fig. 1 (Left) The graph of the cube (Right) All-hexagon toroidal map of the cube graph exhibiting D_{4h} symmetry

Table 1 Rotation scheme of the toroidal map of the cube graph depicted in Fig. 1 Its linear code equals (2,2,2,2,2,2,2,2)

1.	2	5	4	5.	1	8	6
2.	1	6	3	6.	2	7	5
3.	2	7	4	7.	3	8	6
4.	1	8	3	8.	4	7	5

where $deg(v)$ denotes the degree or valency of vertex v. As an example in Fig. 1 we show the graph of the cube and one of its maps on the torus. This toroidal map corresponds to the set of local rotations given in Table 1 (we have chosen an anti-clockwise rotation). Notice that instead of listing the consecutive edges, it is sufficient to list their end-vertices. The set of local rotations, one for each vertex, is denoted as a *rotation scheme*. It is obvious that every oriented map can be described by such a rotation scheme. However, the reverse is also true, namely, every rotation scheme corresponds to an oriented map [5–7]. The problem of finding all maps can thus be reformulated as the problem of finding all rotation schemes. As a rotation scheme is completely defined by the local rotations at all of its vertices, the number of different rotation schemes (= the number of oriented maps) equals:

$$N_r = \prod_{v \in V} n_r(v) \tag{2}$$

By using rotation schemes, the generation of all possible maps has become an easy task, but its reduction to the set of symmetry-distinct maps still poses some serious problems. It involves the very time consuming step of checking for isomorphisms between each newly derived rotation scheme and all previously found symmetry-distinct solutions. A second, although subordinate problem, is related to the storage as the resulting rotation schemes need to be stored as matrices.

3 From rotation scheme to linear code

It is not at all difficult to see how we can turn a rotation scheme into a linear code. From Eq. 1 we know that the number of different local rotations at a given vertex v corresponds with $(deg(v) - 1)!$, which is nothing else than the number of ordered

Table 2 Procedure which connects the six possible vertex-rotations of a tetravalent vertex to their corresponding rotation-states

$v.$	3	4	6	8	$\longrightarrow$	(1234)	"1"
$v.$	3	4	8	6	$\longrightarrow$	(1243)	"2"
$v.$	3	6	4	8	$\longrightarrow$	(1324)	"3"
$v.$	3	6	8	4	$\longrightarrow$	(1342)	"4"
$v.$	3	8	4	6	$\longrightarrow$	(1423)	"5"
$v.$	3	8	6	4	$\longrightarrow$	(1432)	"6"

sequences of edges around v, defined up to cyclic permutations. The problem that a vertex rotation is only defined up to cyclic permutations should not worry us as it can easily be circumvented if we only consider the cyclic permutation which starts with the lowest-labeled vertex. As an illustrating example of this linearization process we take the toroidal map at the right-hand side of Fig. 1. Its corresponding rotation scheme in Table 1 indeed shows that all vertex-rotations start with their lowest-labeled vertex. Because all vertices are trivalent, they can only have two possible vertex-rotations which we will denote as rotation-states "1" and "2". By definition we always let state "1" coincide with the vertex-rotation where the adjacent vertices are ordered in a strictly increasing way. The second state "2" just corresponds with the alternative and inverse rotation. From the rotation scheme in Table 1 we see that all eight vertex-rotations are not strictly increasing so the corresponding linear code of this map reads (2,2,2,2,2,2,2,2). In a similar fashion, there will be a one-to-one correspondence between every possible rotation scheme and the set of linear codes ranging from (1,1,1,1,1,1,1,1) to (2,2,2,2,2,2,2,2). This one-to-one correspondence also makes it possible to retrieve the corresponding rotation scheme from a given linear code. For this purpose one should only have a complete knowledge of the adjacency list of the investigated graph. The generation process of finding all maps can thus simply be reduced to the generation of all possible linear codes. By convention we always start from the smallest possible code which only consists of state "1" rotations, in this case (1,1,1,1,1,1,1,1). To get the next linear code we identify the outermost right location in the present code for which the state can still be increased and raise its value by one. However, if this location does not correspond with the outer right position, all rotation-states located further to the right should be reset to "1". For the present example this just means that the next codes in line will correspond to: (1,1,1,1,1,1,1,2), (1,1,1,1,1,1,2,1), (1,1,1,1,1,1,2,2), (1,1,1,1,1,2,1,1), etc. The process automatically stops when there is no location left which can be further increased, so in our case with the code (2,2,2,2,2,2,2,2). If one replaces the numbers 1 and 2 in these linear codes by, respectively 0 and 1 it can easily be seen that for this trivalent case the generation process just corresponds with listing in increasing order all binary numbers on eight elements.

Extension of this formalism for trivalent graphs to graphs with vertices of higher degree is straightforward. For a tetravalent vertex, for instance, one has six possible vertex-rotations and they can therefore be identified with the *rotation-states* "1" to "6". The link between the exact vertex-rotations and these states is illustrated in Table 2 where we give the six different rotations of an arbitrary tetravalent vertex v together with their corresponding states. The procedure is now as follows. In a first step we

appropriately replace the vertex-labels of the vertex-rotations by the numbers 1–4. This is done in a way that the smallest labeled vertex is replaced by 1, the second smallest by 2, etc. This translates the original rotations into one of the six permutations on the labels 1–4. (One only has six permutations as the starting vertex is always identified with the lowest-labeled vertex). It is now possible to assign the states "1" to "6" to these permutations. State "1" just corresponds with the smallest permutation (1234), state "2" to the second smallest permutation (1243), etc. As an example, the vertex-rotation (3,8,4,6) can be identified with permutation (1423) and corresponds with label "5". Further extensions to vertices of degrees 5, 6, or higher can made along the same line. Notice that there is no need for all vertices to have the same degree. But if this is the case, one should be very cautious during the generation of all linear codes as the higher degree vertices have to run over more states than the lower degree ones. For instance, for a vertex of degree 3 one should only vary its possible states from "1" to "2" where the states of a vertex of degree 5 should already be varied from "1" to "24".

4 Symmetry transform of a linear code

Now that we can represent a map as a linear code we can show how we can simplify the procedure of isomorphism testing. For this purpose it is necessary to define the exact action of the graph automorphisms on a given linear code. Starting from a given map, a graph symmetry only reshuffles the vertex-labels and therefore always leads to a symmetry-equivalent map, although in most instances differently labeled. If the labeling is indeed changed, this vertex shuffling will lead to a different rotation scheme and therefore also to a different corresponding linear code. It is this change of the linear code that makes it possible to change the tedious isomorphism testing process of the previous chapter into the more time-friendly search for a minimal code. Simply stated this means that if during the generation process a linear code is formed, which can be lowered by the action of a symmetry operation, a symmetry-equivalent map should have been found before. Consequently, the newly derived code can instantaneously be removed from any further investigation. The crucial step in this process is of course to define how a graph automorphism can work on a linear code and possibly change it. In the following we fully describe this action, first for the easy case of purely trivalent graphs and later for the more general case of graphs with higher and non-fixed vertex degrees.

4.1 Trivalent maps

Before we turn to the question of the direct action of a graph automorphism on a given linear code, we first describe its exact action on the corresponding rotation scheme. As an illustrative example we take the rotation scheme of the toroidal embedding of the previous section (Table 1) and define the action of the following graph automorphism:

$$C_3 \rightarrow (1)(2,4,5)(3,8,6)(7) \tag{3}$$

Table 3 Rotation scheme of the map which results from applying the C_3 symmetry operation of Eq. 3 to the all-hexagon map of Fig. 1

1.	4	2	5	2.	1	6	3
4.	1	3	8	3.	4	7	2
8.	4	7	5	7.	8	6	3
5.	1	6	8	6.	5	7	2

Table 4 Standard form of the rotation scheme of the symmetry transform of the map of Fig. 1 under the C_3 operation of Eq. 3 The linear code of this transformed map reads (2,2,1,1,1,1,2,2)

1.	2	5	4	5.	1	6	8
2.	1	6	3	6.	2	5	7
3.	2	4	7	7.	3	8	6
4.	1	3	8	8.	4	7	5

Table 5 Symmetry transforms of the rotation states "1" and "2" of vertex 5 under the C_3 symmetry operation of Eq. 3

5.	1	6	8	"1"	$\longrightarrow$	2.	1	3	6	"1"
5.	1	8	6	"2"	$\longrightarrow$	2.	1	6	3	"2"

Within the O_h symmetry group of the cube graph this automorphism corresponds with one of the eight three-fold C_3 operations. This vertex-permutation reshuffles the vertices of the original rotation scheme of Table 1 and thereby leads to the new rotation scheme of Table 3. As this table does not correspond to our standards (we always list the cyclic permutation starting with the smallest-labeled vertex and also list the vertex-rotations according to increasing vertex number) we have transformed it into the standard format and show the result in Table 4. From the latter rotation scheme it can be easily seen that under the action of the C_3 symmetry element the original linear code (2,2,2,2,2,2,2,2) is changed into the code (2,2,1,1,1,1,2,2), which corresponds to a smaller binary number. However, if we had to use this method over and over again to check for the effect of a graph symmetry on a linear code we would not get the dramatic gain in computing time we pointed out before. We shall however see that the symmetry transform of a given linear code can be calculated without any reference to the corresponding rotation schemes.

In the symmetry transformation of a linear code, there are two important points one should realize. A first point is that the new rotation around a given vertex v is not determined by the original rotation around this vertex but by the original rotation around the vertex u which is mapped onto v by the investigated automorphism. A second and very important point is that all states of the original code other than the one of vertex u have completely no influence on the resulting state of vertex v. Lets now as an example investigate the new rotation of vertex 2 under the C_3 symmetry of Eq. 3. Under this symmetry, vertex 2 is the image of vertex 5, so it will be the rotation at vertex 5 which determines the new rotation at vertex 2. Vertex 5 can be in two possible states, namely state "1" or state "2". Both these states and their corresponding ordered adjacency lists are given on the left-hand side of Table 5. Their symmetry transforms and corresponding states under the investigated C_3 automorphism are given on the right-hand side. As the states are not changed we can conclude that under this operation the original

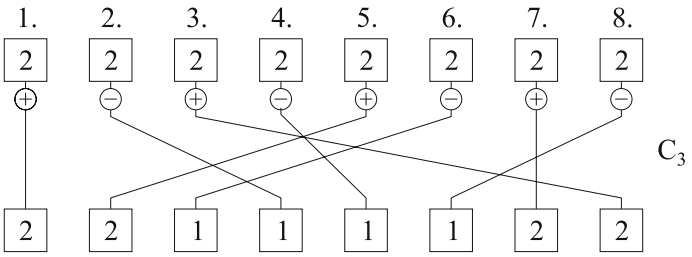

Fig. 2 Schematic overview of the action of the C_3 element of Eq. 3 on the linear code (2,2,2,2,2,2,2,2) of the map of Fig. 1

state of vertex 5 is just transferred to vertex 2. It can however also happen that states are not just transferred but also changed. One such example is the the new state of vertex 3 which by an analysis similar to the one above can be proven to be always opposite to the original state of vertex 6. For trivalent graphs these are the only two possibilities that can occur. As a result, the action of a symmetry operation on a linear code can be represented by means of a double list. The first list indicates from which original vertices the new states should be retrieved and the second list indicates if these states should be kept (+ sign) or reversed (− sign). For the present C_3 symmetry operation we get the following double list (1+,5+,6−,2−,4−,8−,7+,3+). The 6− in third position for instance just means that the new state of vertex 3 is opposite to the state of vertex 6 in the original code. In Fig. 2 we give a schematic overview of this operation on the (2,2,2,2,2,2,2,2) linear code corresponding with the toroidal map of Fig. 1. Here the lines connecting the vertices of the upper and lower code indicate which vertices of the original (upper) code should be used to determine the states of the vertices in the transformed (lower) code. The plus and minus signs on these lines indicate if the states should just be kept or altered during their transfer. The figure easily shows that the transform of the code (2,2,2,2,2,2,2,2) under the investigated C_3 operation will be equal to (2,2,1,1,1,1,2,2). A similar analysis is of course possible for all graph automorphisms and makes it possible to efficiently calculate all symmetry transforms of a given linear code.

4.2 Maps of higher degree

For the trivalent case we only had two possible rotation-states and it was therefore not difficult to describe the action on a given linear code as during the "transfer of states" from vertex u to v the rotation-state could only stay the same or change to its opposite state. For vertices of higher degree, there are more possible states and it is therefore impossible to describe their action by means of a double list. For a tetravalent graph for instance one will need a $6 \times V$ matrix to completely describe the action of a given graph automorphism. To illustrate this we will use a tetravalent map whose corresponding rotation scheme is given in Table 6. The map corresponding with this rotation scheme is shown in Fig. 3, and corresponds with one of the five Platonic solids of genus 3, described in Ref. [8]. The order of the automorphism group of the graph which underlies this map equals 768. Here we list one of its automorphisms which we

Table 6 Rotation scheme of the tetravalent map of Fig. 3 Its linear code reads (6,4,4,4,1,4,6,2,2,2,1,2)

1.	2	12	10	6	"6"	7.	4	12	8	6	"6"
2.	1	9	11	3	"4"	8.	3	5	9	7	"2"
3.	2	8	10	4	"4"	9.	2	4	10	8	"2"
4.	3	7	9	5	"4"	10.	1	3	11	9	"2"
5.	4	6	8	12	"1"	11.	2	6	10	12	"1"
6.	1	7	11	5	"4"	12.	1	5	11	7	"2"

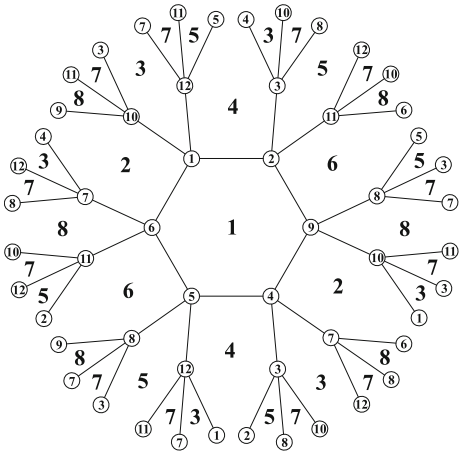

Fig. 3 All-hexagonal Platonic map of genus 3 taken from Ref. [8]

Table 7 Symmetry transform of the rotation scheme of Table 6 under the graph automorphism of Eq. 4 The corresponding linear code reads (1,4,2,4,6,4,1,2,4,2,6,2)

1.	6	10	12	2	"1"	9.	8	10	4	2	"4"
6.	1	7	11	5	"4"	4.	5	3	7	9	"4"
5.	6	4	12	8	"6"	7.	6	8	12	4	"1"
8.	5	9	7	3	"2"	12.	1	5	11	7	"2"
3.	8	2	4	10	"2"	11.	6	2	12	10	"6"
2.	1	9	11	3	"4"	10.	1	3	11	9	"2"

Table 8 Symmetry transforms of the 6 possible rotation-states of vertex 3 under the symmetry operation of Eq. 4

3.	2	4	8	10	"1"	$\longrightarrow$	5.	6	8	4	12	"5"
3.	2	4	10	8	"2"	$\longrightarrow$	5.	6	8	12	4	"1"
3.	2	8	4	10	"3"	$\longrightarrow$	5.	6	4	8	12	"4"
3.	2	8	10	4	"4"	$\longrightarrow$	5.	6	4	12	8	"6"
3.	2	10	4	8	"5"	$\longrightarrow$	5.	6	12	8	4	"2"
3.	2	10	8	4	"6"	$\longrightarrow$	5.	6	12	4	8	"3"

will use to illustrate our procedure:

$$(1)(2,6)(3,5)(4,8)(7,9)(10,12)(11) \tag{4}$$

In Table 7 we show the transformed rotation scheme which results from the action of this graph automorphism on the original map. Notice that we have indicated the

Table 9 The $6 \times V$ transformation matrix which completely describes the action of the graph automorphism of Eq. 4 on all possible linear codes

$$\begin{pmatrix} 6 & 1 & 5 & 6 & \mathbf{2} & 1 & 2 & 6 & 5 & 1 & \mathbf{6} & 1 \\ 4 & 2 & 1 & 4 & 5 & 2 & 5 & \mathbf{4} & \mathbf{1} & \mathbf{2} & 4 & \mathbf{2} \\ 3 & 3 & 4 & 3 & 6 & 3 & 6 & 3 & 4 & 3 & 3 & 3 \\ 2 & \mathbf{4} & \mathbf{6} & \mathbf{2} & 3 & \mathbf{4} & 3 & 2 & 6 & 4 & 2 & 4 \\ 5 & 5 & 2 & 5 & 1 & 5 & 1 & 5 & 2 & 5 & 5 & 5 \\ \mathbf{1} & 6 & 3 & 1 & 4 & 6 & \mathbf{4} & 1 & 3 & 6 & 1 & 6 \end{pmatrix}$$

corresponding rotation-states, but have chosen not to put this table in its usual standard format as this will facilitate our next discussion. As for the trivalent case, the rotation-state of a vertex v is solely determined by the original state of the vertex u which is mapped onto v under the graph automorphism. However, vertex u can have six different states which can all be altered during their transfer to vertex v. It is therefore necessary to calculate the resulting images for each of these six initial states. As an example, in Table 8 we have calculated the images of all six possible rotations of vertex 3 under the operation of Eq. 4 and also derived their corresponding states. The same procedure can be applied to every vertex and leads to the $6 \times V$ matrix given in Table 9. Here the rows denote the values of the initial states (1–6) and the columns correspond with the vertex labels (1–12). The third column therefore exactly corresponds with the results of Table 8. The table can now be interpreted as follows: i.e. the third entry of the fourth row which reads 6 tells us that if the rotation-state of vertex 3 (= column) is equal to "4" (= row) in the original code, it must be changed into rotation-state "6" before is transferred to vertex 5 (= the symmetry transform of vertex 3) of the new code. Notice that every column contains exactly once the states "1" to "6". This is obvious as it can not be that two different rotations of a given vertex are transformed into the same final state. The entries which are relevant for our example are indicated in bold. If we read them from the left to the right we get "1,4,6,2,2,4,4,4,1,2,6,2". One should however be careful at this point as this set does not correspond with the new linear code. It corresponds to the rotation-states of the vertices in the order that they are listed in Table 7 namely 1,6,5,8,3,2,9,4,7,12,11,10. However, if we sort this list of vertices and their corresponding states in increasing order we end up with the transformed linear code which reads (1,4,2,4,6,4,1,2,4,2,6,2). Although it looks like this process will take a lot of computing time, one should realize that the *transformation matrices* have to be calculated only once and from then on can be used to calculate the symmetry transforms of any given linear code.

A further extension of the procedure to graphs with non-fixed vertex degrees is certainly possible. The symmetry transforms of all vertices under a given automorphism can be calculated in just the same way and the results can be stored in a matrix with dimensions $(\text{maxdeg} - 1)! \times V$ where maxdeg corresponds with the highest vertex degree.

5 Implementation

In Fig. 4 we show a flowchart of our newly derived algorithm to find all symmetry-distinct maps of a given graph. The motor of the program is the module which

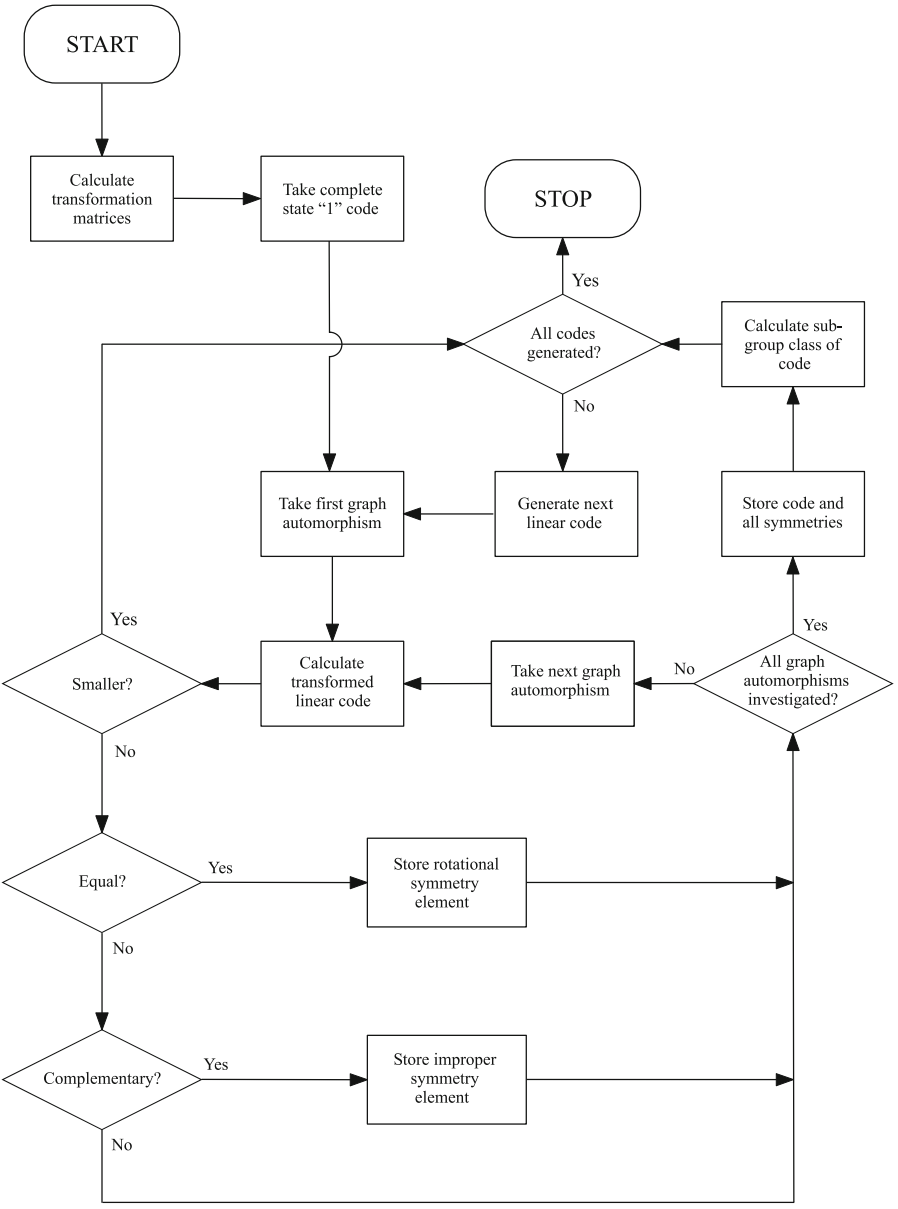

Fig. 4 Flowchart of the improved algorithm to find all symmetry-distinct embeddings

generates all possible linear codes. This module only needs a list of all vertex degrees so that it knows the maximum number of rotation-states for each entry of the linear code. Once a new code is generated it can be transferred to the minimal code module where the actions of all graph symmetries are investigated one by one to see if it is possible to form a smaller code. The only information which is needed at this level

are the transformation matrices which were calculated prior to the actual generation process. The transformed linear code can attain four different forms. A first possibility is that the transformed code is smaller than the original one. In this case we can conclude that a symmetry-equivalent map was already found. We can therefore dismiss the code from any further investigation and directly turn to the generation of the next linear code. If the transformed code is not smaller than the original code we can check if it is equal. If this is the case we have found a graph automorphism which keeps the corresponding rotation scheme unaltered and therefore corresponds with a rotational symmetry element of the map associated with this linear code. In this case we store this symmetry element and directly start investigating the next graph automorphism. If both codes are also not equal we must have a look if they are complementary. A rotation scheme, and therefore also a linear code, is in 1-1 correspondence with maps on oriented surfaces, but in 2-1 correspondence with maps on orientable surfaces [5]. Simply stated this means that if a map is reflexible (= contains orientation-reversing or improper symmetries) its orientation reversed counterpart (although they are symmetry-equivalent) will also be generated by our procedure. In our search for a minimal code this will not pose any problems. However, if we also want to have a complete list of all improper symmetry elements of the map we should be able to recognize its complementary code. The complementary code of a map just corresponds with the code of the map where all vertex-rotations are reversed. For the code of a trivalent map this is an easy process as the states "1" and "2" are each others complement. For vertices of higher degrees one must explicitly calculate all complementary couples. For a tetravalent vertex for instance one can see that the states "1" (1234) and "6" (1432), "2" (1243) and "4" (1342), and "3" (1324) and "5" (1423) are each other complements. The complement of the linear code (6,4,4,4,1,4,6,2,2,2,1,2) of the map of Table 6 therefore reads (1,2,2,2,6,2,1,4,4,4,6,4). If we find that the transformed code is indeed complementary we will add it to the list of improper symmetry elements of the map and start investigating the next graph automorphism. Otherwise the code is just larger than the original one and not special in any way so we directly turn to the next automorphism. If at a certain moment all graph automorphisms are investigated and none of them was able to produce a smaller code, we can conclude that we have found a new map. It can therefore be stored, as a linear code of course, and subjected to further investigation. It is important to realize that at this point the complete symmetry of the map is already known and can be allocated to one of the subgroup classes of the automorphism group of the graph. An algorithm to derive these subgroup classes has been discussed in Ref. [8]. Once the corresponding subgroup class is derived we return to the beginning of our process and generate the following code in line. This process is continued until all linear codes have been generated. The result is a list of all symmetry-distinct maps represented by their minimal linear codes. Note that if one wants further details of the derived maps, such as their genus γ or the sizes and exact structure of their faces, the linear codes have to be translated back into rotation schemes. Because of the 1-1 correspondence between these two representations and our complete knowledge of the graph adjacency list this poses no further problems.

Based on the algorithm above we wrote a Fortran program which can retrieve all symmetry-distinct embeddings of a graph and will only need its adjacency list as input. Before we apply this program to the highly symmetrical Dyck graph and give

its results, we will first briefly describe the efficiency of the algorithm and give some smart implementations which can make the algorithm run even faster.

6 Efficiency of the algorithm

Although the isomorphism checking procedure is greatly improved by the linear code method, it still remains the most time-consuming step of the algorithm. Any further improvements on this procedure can therefore lead to a substantial gain in computing time. As the list of graph automorphisms is randomly ordered, in case of a duplicate solution one has to run on average through half the number of graph automorphisms before one finds a smaller code. This can however be a serious task, especially when the automorphism group of the graph is quite large. The question which naturally arises is therefore if there exists an efficient ordering of the set of graph automorphisms which fastens up this process. In other words, is there an ordering such that we have to run on average through less than half the amount of automorphisms. Well the answer is yes and no at the same time. It is no in case one searches for one single ordering that can be used during the whole generation process. However, the answer is yes if one allows a dynamic ordering of the automorphisms. With dynamic we mean that the ordering is not fixed from the beginning to the end, but varies according to the needs of the algorithm at a certain time. The possible improvements of such a dynamic ordering are based on the following two observations:

1. The transformed state of a given vertex v is solely determined by the original state of the vertex u which is mapped onto v by the automorphism under consideration.
2. During the generation process, the states of positions located at the beginning of the linear code vary slower than those located at the end.

To see how such a dynamic ordering can help us, just consider the following imaginary linear code which starts with (3,5,4,5,…) together with a graph automorphism which permutes vertex 4 to vertex 1 and changes its state from “5” to “2”. This directly implies that all linear codes starting with this sequence will be lowered by this automorphism. So, if we could place this automorphism at the beginning of our list for as long as this initial sequence is not changed, it would lead to a considerable speeding up as the first encountered automorphism would directly lead to a smaller code. Of course this effect is only temporary as from the moment that the state of vertex 4 changes into state “6” it can well be that its image will be higher than “3” (the state of the first vertex of the code) and therefore no longer leads to a smaller code. If this is the case, we end up in the worst case scenario for as long as this vertex 4 stays in state “6” it will always give rise to higher codes. From here on this automorphism should therefore best be moved to the end of the list. The previous discussion is quite intuitive, but how can we incorporate these observations into our computer code? One way to do this is to keep track which automorphisms have lowered the most recently investigated duplicates. For this purpose we need to use a stack which has a length equal to the order of the graph automorphism group and for which each entry corresponds to exactly one of the graph automorphisms. Initially, all entries of this stack are set to zero. Once the generation of all linear codes has started, each time a duplicate is found we identify the automorphism which has lowered its linear code and increase its corresponding stack

entry by one. After a certain number of linear codes have been investigated (let us say 10000 but this number can be changed at will) the algorithm takes a look at the stack to see which automorphisms have lowered the previous 10000 codes most of the time. Based on the previously discussed observations we can assume that they also have a great likelihood to lower the next set of 10000 codes. We therefore update our list of automorphisms and put the most used automorphism at the beginning and the least used at the end. Of course at this point we should reset all stack-values back to zero. Otherwise we would drag all previous results with us through the whole algorithm. On the contrary, as to fully exploit the advantages of this dynamic ordering one should only focus on the most recent results. The same analysis can be made after the next interval of 10000 codes and so on.

The efficiency of our algorithm is of course determined by the average number of automorphisms one has to investigate before a smaller linear code is found. In the best case scenario, one disposes of an automorphism list which is ideally ordered at any time of the algorithm. This simply implies that each duplicate code will be lowered by the first listed automorphism. One should however be careful as this lower bound of one investigated automorphism is not theoretically possible. The reason is that each time we find a new symmetry-distinct map we have to run through all possible automorphisms, $|Aut(\text{graph})|$, as its corresponding linear code can not be lowered. The remaining $(|Aut(\text{graph})|/|Aut^{+}(\text{map})|) - 1$ duplicates ($|Aut^{+}(\text{map})|$ stands for the order of the rotational subgroup of the map) which are characterized by higher linear codes can however theoretically be lowered by the first automorphism. One can therefore state that the total number of automorphisms which have to be checked for a symmetry-distinct map with a rotational subgroup of order $|Aut^{+}(\text{map})|$ equals:

$$|Aut(\text{graph})| + (|Aut(\text{graph})|/|Aut^{+}(\text{map})|) - 1 \tag{5}$$

The average number of automorphisms which have to be checked before a smaller code is found, is therefore given by:

$$\frac{\sum[|Aut(\text{graph})| + (|Aut(\text{graph})|/|Aut^{+}(\text{map})|) - 1]}{\#\text{linear codes}} \tag{6}$$

where the summation is over all symmetry-distinct maps. The formula above gives an exact theoretical lower bound and will be different for each investigated graph. However, if we assume that all maps have the trivial C_1 symmetry (which is not implausible as for larger and larger graphs the fraction of maps with C_1 symmetry tends asymptotically to one) we find from Eq. 5 that the total number of automorphisms which have to be checked equals:

$$2|Aut(\text{graph})| - 1 \tag{7}$$

A map with C_1 symmetry will be generated exactly $|Aut(\text{graph})|$ times by the algorithm so the average number of automorphisms which are needed to find a lower code equals:

$$\frac{2|Aut(\text{graph})| - 1}{|Aut(\text{graph})|} \tag{8}$$

which for large graph automorphism groups tends to 2. The efficiency of our algorithm can therefore be tested by comparing the average number of tested automorphisms for each linear code with this theoretical lower bound or otherwise stated: *the closer this number gets to two the more efficient our calculation will be*. However, for smaller graphs the fraction of maps with symmetries other than C_1 will be significant and the theoretical limit will be higher than 2. One therefore has to use the general formula of Eq. 6 to check for efficiency. For the case of the cube graph, for instance, the theoretical lower bound will be equal to 3.570. In Chap. 8 we investigate the case of the Dyck graph which will be large enough to check for the real efficiency of our algorithm.

7 Correctness of the algorithm

If one implements a new algorithm, one of course needs some procedures to check its correctness. An obvious way is to compare the results of our algorithm to cases for which the complete set of symmetry-distinct solutions is known. Within the mathematical literature however the emphasis is mainly on the enumeration and not on the generation of all symmetry-distinct solutions. The problem of enumerating all orientable symmetry-distinct embeddings for a given graph was not solved analytically until 1988 when Mull et al. derived a general enumeration method based on the famous *Burnside lemma* [9]. For the present context the Burnside lemma reads:

$$|C(G)| = \frac{1}{\text{Aut } G} \sum_{\alpha \in \text{Aut } G} |F(\alpha)| \tag{9}$$

where $|F(\alpha)| = \{\rho \in R(G) | \alpha(\rho) = \rho\}$. In this formula $|C(G)|$ denotes the number of symmetry-distinct embeddings, $|\text{Aut } G|$ the order of the automorphism group of the graph and α one of the graph automorphisms. ρ corresponds with a given rotation scheme and $R(G)$ stands for the set of all rotation schemes of the graph. The most important symbol $|F(\alpha)|$ is called the fixed set of α and corresponds with the number of rotation schemes which are fixed (remain the same) under the action of α. The analytical calculation of these fixed sets is thoroughly described in Ref. [9]. To calculate them one only needs a complete knowledge of the adjacency list of the graph and the exact structure of one automorphism out of each conjugacy class of the graph automorphism group. We are however not going into the mathematical details but only state that we used the results of the paper of Mull et al. to write a Fortran program which can analytically calculate the number of symmetry-distinct embeddings for a given graph. Using this program we could check that the number of symmetry-distinct solutions produced by our algorithm is indeed correct.

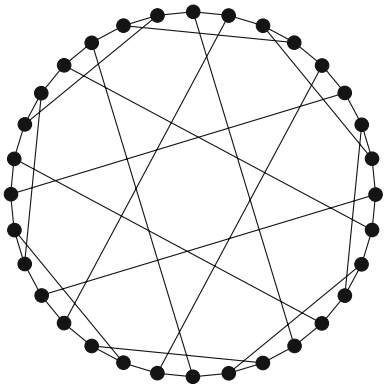

Fig. 5 The highly symmetrical 32-vertex Dyck graph

Table 10 Listing of all symmetry-distinct maps of the Dyck graph according to their genus and the order of their rotational symmetry group

Order →	192	96	64	48	32	24	16	12	8	6	4	3	2	1	Total
$\gamma = 1$	–	1	–	–	–	–	–	–	–	–	–	–	–	–	1
$\gamma = 2$	–	–	–	–	–	–	–	–	–	–	–	1	1	–	2
$\gamma = 3$	–	1	–	–	–	–	1	–	1	3	3	4	11	26	50
$\gamma = 4$	–	–	–	–	–	–	–	1	5	2	6	21	99	1,190	1,324
$\gamma = 5$	–	–	–	–	2	2	1	1	12	6	37	98	812	51,742	52,713
$\gamma = 6$	–	–	–	–	–	–	–	–	1	23	6	234	4,888	1,330,264	1,335,416
$\gamma = 7$	–	–	–	–	–	–	4	–	12	18	115	300	7,515	9,824,060	9,832,024
$\gamma = 8$	–	–	–	–	–	–	–	–	2	8	29	332	7,529	11,151,518	11,159,418
Total	–	2	–	–	2	2	6	2	33	60	196	990	20,855	22,358,800	22,380,948

Table 11 Listing of all symmetry-distinct maps of the Dyck graph according to their genus and the order of their full symmetry group

Order →	192	96	64	48	32	24	16	12	8	6	4	3	2	1	Total
$\gamma = 1$	1	–	–	–	–	–	–	–	–	–	–	–	–	–	1
$\gamma = 2$	–	–	–	–	–	–	–	–	–	1	1	–	–	–	2
$\gamma = 3$	1	–	–	–	1	–	1	3	1	4	7	–	18	14	50
$\gamma = 4$	–	–	–	–	–	1	5	2	6	13	37	8	284	968	1,324
$\gamma = 5$	–	–	2	2	1	1	10	6	23	58	196	40	3,324	49,050	52,713
$\gamma = 6$	–	–	–	–	–	–	1	7	6	120	628	130	23,188	1,311,336	1,335,416
$\gamma = 7$	–	–	–	–	4	–	12	10	29	76	579	232	58,672	9,772,410	9,832,024
$\gamma = 8$	–	–	–	–	–	–	–	0	3	14	47	326	8,434	11,150,594	11,159,418
Total	2	0	2	2	6	2	29	28	68	286	1,495	736	93,920	22,284,372	22,380,948

8 Case study: the Dyck graph

The Dyck graph is characterized as the only cubic symmetric graph on 32 vertices. Its exact connectivity is shown in Fig. 5. The graph is highly symmetrical as the order of its automorphism group equals 192. It consist of 14 conjugacy classes and 83 subgroup conjugacy classes. Using the linear code generation algorithm we obtained a complete symmetry classification of all distinct maps according to these subgroup conjugacy classes. The result are shown in Tables 10 and 11, where we list all

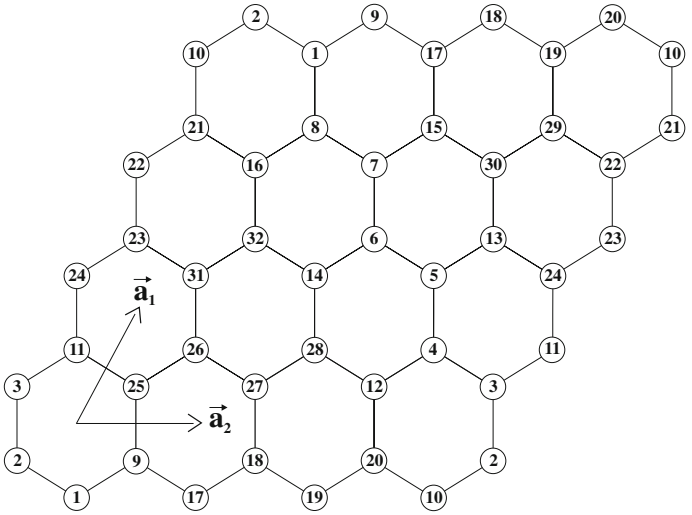

Fig. 6 Highly symmetrical map of the Dyck graph on the torus

symmetry-distinct maps according to their genus and the order of their rotational symmetry group (Table 10) and of their full symmetry group (Table 11). From these tables we directly see that the set of $2^{32} = 4\,294\,967\,296$ maps (see Eq. 2) is reduced to a set of 22 380 948 symmetry-distinct solutions. Our previous statement that for large graphs the greatest fraction of the maps has trivial C_1 symmetry is also confirmed, so we expect the theoretical lower limit to be around two. Using Eq. 6 and the results of Table 10 the exact lower bound can be calculated and equals 1.995. During our calculations the average number of checked automorphisms was 2.506 which is only 26% higher than the theoretical limit and very good if we keep in mind that we came from an algorithm where we needed to check an average of 96 automorphisms (half the group order of the graph) to find a lower code. The total running time of the optimized program for this example was approximately 4 h on an Intel Pentium 4/2.5 GHz processor. This Dyck graph with 32 trivalent vertices therefore lies at the boundary of what is still feasible in reasonable time. A close inspection of Table 11 shows two very interesting maps whose symmetry groups correspond exactly with the full automorphism group of the graph. Their order, 192, is exactly equal to four times the amount of edges, and therefore correspond with a genus-1 and genus-3 regular map [8], where the genus-1 map is not only regular but also minimal. Both structures will be fully described in the next section.

8.1 Molecular realization

From a chemical point of view, the Dyck graph is very interesting [10,11]. It is trivalent and can therefore serve as the blueprint for a sp^2 hybridized carbon allotrope. In order to be chemically relevant the maps should however be realized in 3D space. A first possible 3D realization of the Dyck graph can be based on the all-hexagon map on the torus [12,13], shown in Fig. 6. Because of its very small dimensions it will however

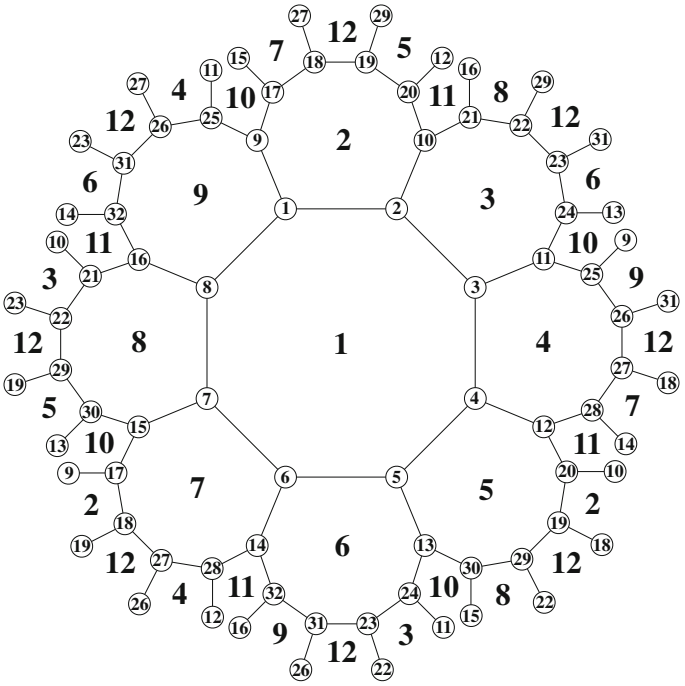

Fig. 7 Highly symmetrical map of the Dyck graph on the triple torus

incorporate a serieus amount of strain and therefore it does not lead to a chemically plausible structure. In principle, the strain of the toroidal closing can be easily removed if one opens up the torus to form a small cylinder. Connecting several of the cylinders together leads to a nanotube-like structure, which we might characterize as a (4,0) zigzag tube, but even such a tube would probably not be very realistic as its diameter will be to small.

We are therefore more interested in the alternative highly symmetrical genus-3 map where, instead of the hexagons, the octagons of Fig. 7 become faces [11]. This map is deeply connected with the previous one by a process called Petrie-Dualization [14]. During this dualization process the symmetry of the original map and its underlying graph are retained, but the topology of the map can change as the faces of the map are replaced by a new set of faces, formed by taking Petrie-paths. A Petrie-path is defined as a path where alternatively left and right turns are taken. As a consequence three consecutive edges of such a path will never belong to the same parental face. In our case, the Petrie paths of the genus-1 map shown in Fig. 6 can be identified with the zigzag paths in the directions $\vec{a}_1$, $\vec{a}_2$ and $\vec{a}_1 - \vec{a}_2$, and correspond exactly with the 12 octagonal faces of Fig. 7.

The realization of this map on a closed genus-3 surface once again leads to an access of strain. The better option therefore consists in finding a realization on a negatively curved open structure, thereby removing most of its strain. In Fig. 8 we show the most symmetrical 3D-realization of this map on the Plumber's nightmare surface (The exact

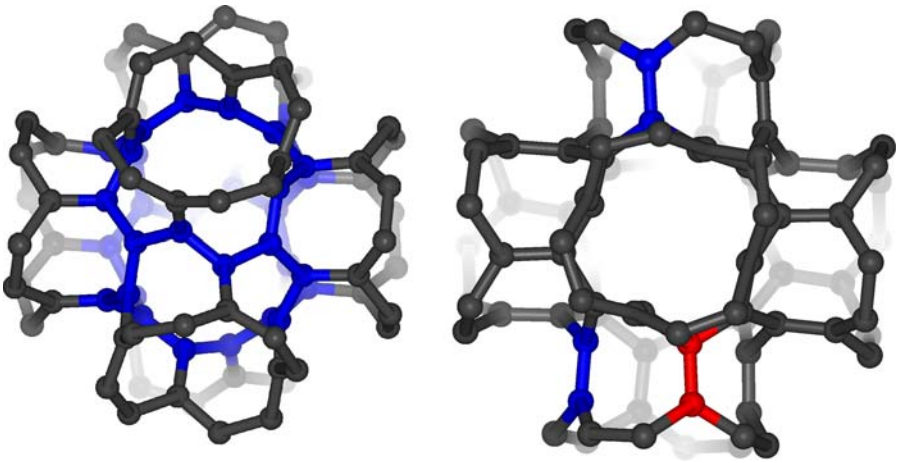

Fig. 8 Embedding of the Dyck graph on the Plumber's nightmare surface. Left: Atoms and edges indicated in blue belong to the same unit-cell; all other vertices belong to neighboring unit-cells and are only included to show the octagonal faces. Right: Equivalent edges are indicated in blue and lead to a 60° twist on identification. Identification of the upper blue and lower red edge leads to the least strained structure which still incorporates a twist of 30°

procedure behind this process can be found in Ref. [5]), which easily lends itself to propagation in all three spatial directions, thereby forming a zeolite-type structure. Notice that only the blue atoms belong to the actual unit-cell. All other vertices belong to neighboring units but are included to show the octagonal faces, which are distributed between neighboring subunits.

If one wants to construct a network that completely mimics the original adjacencies of the Dyck graph, one should identify the blue edges on the right-hand side of Fig. 8. As can be seen from this figure, these edges are not properly located to make this match by simple periodic translation, but instead they are twisted over an angle of 60° [15]. The same mismatch exists when the connections are made in the other two spatial directions. On the left-hand side of Fig. 9 we show a $2 \times 2 \times 2$ supercell formed by making these connections which preserve the Dyck adjacencies and optimized the resulting structure by MM+ molecular modeling. As a result of the 60° twists, the shapes of the individual unit-cells are quite distorted from the relaxed structure of Fig. 8, which means that the structure exhibits considerable strain. Aside from this structure, an alternative and less strained allotrope can be found if one identifies the upper blue edge with the lower red edge. The mismatch between the edges is hereby reduced from 60° to 30°. However, one should keep in mind that by making this new identifications, one changes the connectivity of the decorating graph. As in the previous case, a $2 \times 2 \times 2$ supercell was constructed and optimized using MM+. The resulting structure is shown on the right-hand side of Fig. 9. The figure shows that in this case the individual plumber-subunits are less distorted from their original shape. The 30° structure will therefore exhibit less strain and makes a more plausible chemical candidate than the 60° all-Dyck structure.

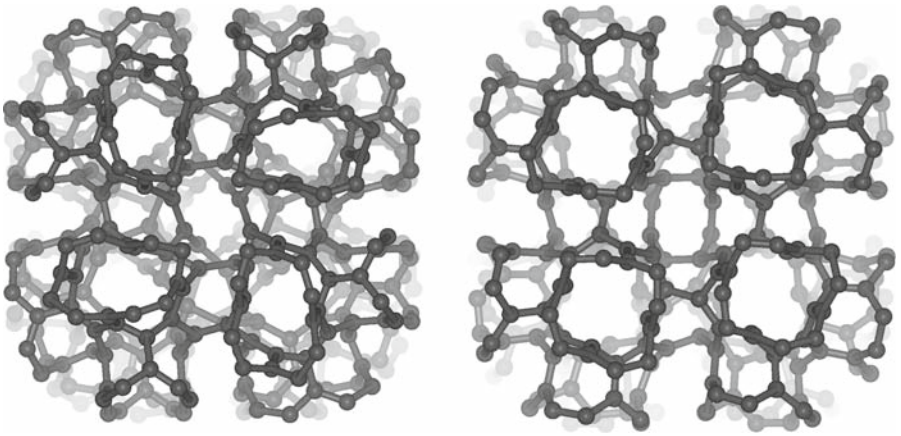

Fig. 9 Supercells formed by introducing twists of 60° (left) and 30° (right) between neighboring subunits (Taken form Ref. [15]). The left structure preserves the connectivity of the original Dyck graph

9 Conclusions

By representing maps as linear codes instead of rotation schemes, we showed that the problem of finding all symmetry-distinct maps can be solved much more efficiently. The reason is that one no longer has to check for isomorphism between the newly derived map and previously stored solutions but instead can use the "minimal code method" which makes it possible to discard all linear codes which can be transformed into smaller codes. For this purpose one only has to define the actions of the graph automorphisms on the linear codes. We proved that these actions can easily be calculated and stored in the form of a set of transformation matrices. As these matrices stay completely invariant, they should only be calculated once, and from then on can be used during the whole generation process. Some further improvements are possible if we allow our list of graph automorphisms to be sorted dynamically according to the needs of the algorithm at a specific time. Using this optimized algorithm we calculated all symmetry-distinct maps of the Dyck graph and showed that our implementation comes quite close to achieving the theoretical perfection. The two most symmetrical maps of this Dyck graph are identified as a genus-1 and genus-3 regular map. The latter forms a blueprint for a interesting negatively curved carbon allotrope solely composed of octagons.

Acknowledgement E. Lijnen holds a post-doctoral fellowship from the Fund for Scientific Research—Flanders.

References

1. A.T. Balaban, *Chemical Applications of Graph Theory* (Academic Press, London, 1976)
2. B. Mohar, C. Thomassen, *Graphs on Surfaces* (Johns Hopkins University Press, Baltimore, 2001)
3. J.L. Gross, T.W. Tucker, *Topological Graph Theory* (Dover Publications, New York, 2001)

4. A. Ceulemans, E. Lijnen, The polyhedral state of molecular matter. Eur. J. Inorg. Chem. **7**, 1571–1581 (2002)
5. E. Lijnen, A. Ceulemans, Oriented 2-cell embeddings of a graph and their symmetry classification: generating algorithms and case study of the Möbius-Kantor graph. J. Chem. Inf. Comput. Sci. **44**, 1552–1564 (2004)
6. J. Edmonds, A combinatorial representation for polyhedral surfaces. Am. Math. Soc. Not. **7**, 646 (1960)
7. In Ref. 5 we describe how the original topological map can be retrieved from a rotation scheme by a procedure called face-tracking
8. E. Lijnen, A. Ceulemans, Topology-aided molecular design: the Platonic molecules of genera 0 to 3. J. Chem. Inf. Model. **45**(6), 1719–1726 (2005)
9. B.P. Mull, R.G. Rieper, A.T. White, Enumerating 2-cell imbeddings of connected graphs. Proc. Am. Math. Soc. **103**, 321–330 (1988)
10. R.B. King, Negative curvature surfaces in chemical structures. J. Chem. Inf. Comput. Sci. **38**, 180–188 (1998)
11. A. Ceulemans, E. Lijnen, L.J. Ceulemans, P.W. Fowler, The tetrakisoctahedral group of the Dyck graph and its molecular realization. Mol. Phys. **102**, 1149–1163 (2004)
12. D. Marušič, T. Pisanski, Symmetries of hexagonal molecular graphs on the torus. Croat. Chem. Acta **73**, 969–981 (2000)
13. K. Kutnar, A. Malnič, D. Marušič, Chirality of toroidal molecular graphs. J. Chem. Inf. Model. **45**, 1527–1535 (2005)
14. P. McMullen, E. Schulte, *Abstract Regular Polytopes* (Cambridge University Press, Cambridge, 2002)
15. E. Lijnen, A. Ceulemans, The symmetry of the Dyck graph: group structure and molecular realization. in *Nanostructures: Novel Architecture*, ed. by M. Diudea (Nova Publishers, New York, 2005)