MATCH
Communications in Mathematical and in Computer Chemistry

MATCH Commun. Math. Comput. Chem. **78** (2017) 713-736

ISSN 0340 - 6253

A Generation Algorithm for Nanojoins

Gunnar Brinkmann, Dieter Mourisse

Applied Mathematics, Computer Science and Statistics
Ghent University Krijgslaan 281-S9
9000 Ghent, Belgium

Gunnar.Brinkmann@UGent.be , Dieter.Mourisse@UGent.be

(Received April 19, 2017)

Abstract

Nanojoins or *nanojunctions* are substructures of large Carbon molecules that connect two or more nanotubes with the same or different tube parameters. Except in the trivial case where two tubes with the same parameters are connected, nanojoins also contain one or more heptagons in addition to hexagons and possibly pentagons like they also occur in fullerenes and nanotubes.

We consider nanojoins as isomorphic when the infinite molecules given by the join connected to infinite tubes are isomorphic. Even when fixing the number of tubes to be connected, the tube parameters, and the number of pentagons and heptagons, there are in most cases infinitely many non-isomorphic nanojoins, so complete enumeration is impossible unless also a parameter restricting the size of the join is added. In this article we describe an algorithm for complete enumeration of non-isomorphic nanojoins for a given set of tube parameters, number of heptagons, and an upper bound on the number of faces in the join. We also give performance statistics and results for some parameter sets.

1 Introduction

The study of fullerenes is an important part within the field of chemistry. In 1996, the Nobel prize for chemistry was awarded for the discovery of the Buckminsterfullerene (also referred to as the buckyball) by Robert Curl, Richard Smalley and Sir Harry Kroto.

Fullerenes are spherical carbon molecules in which all atoms are carbon atoms and every atom is connected to three other atoms. All atoms occur in rings of pentagons or hexagons. The structure of a fullerene can hence be modelled as a 3-regular graph on the surface of a sphere or – equivalently – in the plane, where all faces are pentagons or hexagons. As an easy consequence of the Euler formula, every fullerene has exactly

12 pentagons. Various algorithms to generate complete lists of fullerenes with a given number of atoms have been proposed. The fastest generator at the moment is described in [2].

Nanotubes are a subclass of fullerenes. Nanotubes contain two areas where 6 pentagons are relatively close to each other – called nanocaps – connected by a long tube of only hexagons. The structure of the tube can be described by two parameters (l, m) and is determined by the caps [5]. For $l \neq m$ and $l, m \neq 0$ the parameters (l, m) and (m, l) describe different tubes – or to be exact: tubes that are mirror images of each other. for $l = m$, $m = 0$ or $l = 0$ the parameters (l, m) and (m, l) describe the same tubes and are considered to be identical parameters. As the tubes are very long compared to the small caps, also one side infinite nanotubes are considered – that is caps together with a one side infinite tube of hexagons. Also for nanocaps and one side infinite nanotubes generators have been developed. They generate complete lists of nanocaps and one side infinite nanotubes for given tube parameters. The fastest generator at the moment is described in [4].

A *half-tube* is a one side infinite nanotube with the cap removed. For a tube or half-tube of only hexagons the tube parameters (l, m) – and with them also some physical properties – are uniquely determined, but for some applications (see [10]) it is necessary to combine half-tubes with different parameter sets in one molecule and even molecules with more than two halftubes have been observed in experiment [7]. In such molecules substructures called *nanojoins* (or nanojunctions) containing heptagons, and in most cases also hexagons and possibly also pentagons are involved. Nanojoins can join two or more half-tubes with the same or different parameters.

Though nanojoins are known to be potentially important for applications and have also been observed in nature, until now no generators for complete classes of nanojoins for given parameters were developed and only examples without any claim of completeness have been published (see e.g. [8], [11], [12]).

It is an easy consequence of the Euler formula (see e.g. [3]) that a nanojoin joining k half-tubes must have $p + 6(k - 2)$ heptagons if it contains p pentagons.

Constructions and theorems in [3] give the following theorem:

Theorem 1.1. *For any $k \geq 3$ parameter sets $(l_1, m_1), \ldots, (l_k, m_k)$, any $p \geq 0$ and $h = p+6(k-2)$ an infinite number of non-isomorphic infinite plane graphs containing exactly k*

half-tubes with parameters $(l_1, m_1), \ldots, (l_k, m_k)$*, p pentagons and* $h = p+6(k-2)$ *heptagons exist. For* $k = 2$ *half-tubes there are also infinitely many such graphs if* $p \geq 2$*, no graphs at all if* $(l_1, m_1) = (l_2, m_2)$ *and* $p = s = 1$ *and at least one graph if* $(l_1, m_1) \neq (l_2, m_2)$ *and* $p = h = 1$*.*

It is still open whether for $p = h = 1$ and two parameter sets $(l_1, m_1) \neq (l_2, m_2)$ the infinite plane graphs are uniquely determined.

We will describe an algorithm for complete enumeration of all nanojoins for given $(l_1, m_1), \ldots, (l_k, m_k)$, $p \geq 0$ and $h = p + 6(k-2)$ where all non-hexagons are contained in a substructure with at most f faces. Without the additional parameter describing the locality of the non-hexagon faces complete enumeration is not possible as (see Theorem 1.1) in most cases there are infinitely many nanojoins.

2 Preliminaries

Nanojoins are finite substructures of infinite plane graphs. We will model the edges connecting it to the infinite rest by *semi-edges* which start at a vertex v but have no endpoint. We denote these edges as singletons $\{v\}$ and call an edge e with two endpoints a *complete edge* in cases where we want to emphasize that e is not a semi-edge. If we talk about the size of a face, we always mean the number of vertices in the boundary, so semi-edges do not contribute to the size of a face.

Let $G = (V, E)$ be a simple graph with vertex set V and edge set $E \subseteq \binom{V}{2} \cup \binom{V}{1}$.

We denote the degree of a vertex v not counting semi-edges as $\deg(v)$ and the *full degree* also counting semi-edges as $\overline{\deg}(v)$.

In a nanojoin there can be at most one semi-edge at a vertex v. A semi-edge $\{v\}$ will be represented in figures by a short line segment with one end at vertex v. See, for example, Figure 1.

For various constructions it will be useful to orient the complete edges and interpret each undirected edge as two oppositely directed edges. We write $[u, v]$ for the edge $\{u, v\}$ oriented from u to v. Semi-edges $\{v\}$ correspond to only one directed edge $[v]$.

Definition 1. A *face* in a plane graph is a sequence $[v_0, v_1], [v_1, v_2], \ldots, [v_k, v_0]$ of directed edges so that for $0 \leq i \leq k$ there are no complete edges in clockwise order between $[v_i, v_{(i-1) \bmod (k+1)}]$ and $[v_i, v_{(i+1) \bmod (k+1)}]$. If there is a semi-edge $[v_i]$ between

$[v_i, v_{(i-1) \bmod (k+1)}]$ and $[v_i, v_{(i+1) \bmod (k+1)}]$ we say that this edge belongs to f. Informally speaking such a cycle represents the face on its left.

The *degree sequence* starting at v_0 of a face $F = [v_0, v_1], [v_1, v_2], \ldots, [v_k, v_0]$ (we write $\text{border}(F) = v_0, v_1, \ldots, v_n$) is the string

$$\deg(v_0), \deg(v_1), \ldots, \deg(v_n).$$

If no explicit starting vertex is given, the string is interpreted as a cyclic sequence.

With these notations we can now give a formal definition of a nanojoin. Note that we require that two faces to which halftubes can be attached do not share an edge, but are separated by pentagons, hexagons or heptagons.

Definition 2. A *connecting face* in a graph is a face with a degree sequence that can be written as $(3,2)^l(2,3)^m$ for some $l, m \geq 0$. We sometimes abbreviate such degree sequences as (l, m).

A *nanojoin* is a plane graph with the following properties:

- all vertices v have full degree $\overline{\deg}(v) = 3$
- all faces are pentagons, hexagons, heptagons or connecting faces
- there are at least two connecting faces
- all semi-edges are contained in connecting faces
- no two connecting faces share an edge

It turned out that for the generation of nanojoins a combination of a bottom up and a top down approach performed best for most parameter combinations. We will first describe the bottom up part of this combined approach.

3 Bottom up part

In this part, we will not generate nanojoins, but simply connected patches that can serve as building blocks to form nanojoins in the top down part. We will focus on patches with at most 3 pentagons. An extension of this algorithm that can also generate nanojoins was developed in [9]. This extension uses additional operations that allow the construction

of patches and joins with more than 3 pentagons, but in many cases slow down the generation when applied. In combination with the top down part it turned out to be faster to restrict the bottom up part – that will only be used to develop bounding criteria – to at most 3 pentagons. This restriction is only for this part and has no impact on the number of pentagons allowed in the combined approach. In this article we will use the bottom up approach described in [9] only as a fairly independent check of the results and not describe all details.

Definition 3.

- A patch is a 2-connected plane graph where all vertices have full degree 3, all faces except one (the outer face) are pentagons, hexagons or heptagons and all semi-edges are in the outer face.
- A pseudo-convex patch is a patch where no two vertices without semi-edges share an edge in the boundary.
- A *marked* pseudo-convex patch (short mpc-patch) is a pseudo-convex patch P together with a vertex v in the outer face of P so that the degree sequence of the outer face starting at v is lexicographically maximal. Unless the patch is trivial – that is: a single face – lexicographically maximal sequences always start at a vertex without semi-edges, but there may be more than one vertex from which the sequence is maximal.

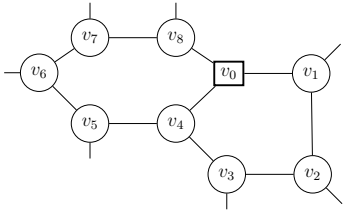


Figure 1. A marked pseudo-convex patch. The vertex with the mark is drawn as a square.

Lemma 3.1. *If a patch has p pentagons, h heptagons and the number of vertices in the boundary with degree 2, resp. 3 is denoted as n_2, resp. n_3, then*

$$n_2 - n_3 = h + 6 - p$$

.

Proof. The outer face has size $n_2 + n_3$ and we denote the number of hexagons by s. By summing up the sizes of all faces we count every complete edge twice, so with E_c the set of complete edges we get

$$2|E_c| = 7 * h + 6 * s + 5 * p + n_2 + n_3$$

Each vertex has full degree 3. By taking semi-edges into account we get

$$3|V| = 2|E_c| + n_2$$

Finally the number of faces $|F|$ is

$$|F| = p + s + h + 1$$

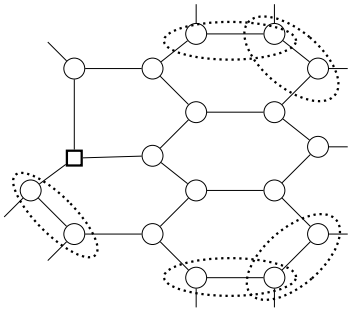
Inserting these formulas into the Euler formula $|V| - |E_c| + |F| = 2$ for the plane we get the desired result. □

Due to this lemma mpc-patches with at most 5 pentagons have more vertices with degree 2 in the boundary than vertices with degree 3. So they have at least one position with two neighbouring vertices of degree 2. As they are pseudo-convex, between two such positions the degrees are alternately 3 and 2 and we can code the boundary sequence uniquely in a shorter way by counting the vertices of degree three between the pairs of vertices of degree two.

Definition 4. If the cyclic boundary sequence of an mpc-patch P can be written as $2, (2,3)^{k_1}, 2, (2,3)^{k_2}, 2, \ldots, (2,3)^{k_l}$ with $k_i \in \mathbb{N}$ for $1 \leq i \leq l$, then the cyclic sequence $(k_1, k_2, \ldots, k_l)$ is called the border code of P.

Fixing a pair of neighbouring vertices of degree 2 we have a string again and can equivalently define the position of the mark as on a first vertex of degree 3 in a part of the boundary corresponding to the first entry in a lexicographically maximal (non-cyclic) border code.

An example for the border code is given in Figure 2.



$2, 0, 2, 0, 1$

Figure 2. The border code of an mpc-patch

First we will describe a method to generate all non-isomorphic mpc-patches with a given number ($p \leq 3$) of pentagons, hexagons and heptagons.

Definition 5. A cut path in a non-trivial mpc-patch P with mark v is a path that starts at the edge e incident with v that is not in the boundary, ends at another boundary vertex and goes alternately first right and then left.

Theorem 3.2. *Each non-trivial mpc-patch P with at most* 3 *pentagons has a unique cut path.*

Proof. Note that P has at most 3 pentagons.

We start at the interior edge of the mark s_0 and construct a maximal path that goes alternately right and left without meeting another boundary vertex or using a vertex twice. In case the next vertex would be a boundary vertex (which would be different from s_0), we have a cut path. We have to prove that the other case – that is: the next vertex would already be contained in the path – cannot occur.

Suppose that we want to add an edge $[s_n, w]$ to the path $S = s_0, s_1, \ldots, s_n$ and $w = s_i$, which implies $i > 0$. Then $S = s_i, s_{i+1}, \ldots, s_n$ is a cycle in the patch P and the part that does not contain the outer face is a patch P' with boundary S. As the vertices from s_{i+1} to s_n have alternating degrees, we conclude that in this part, the number of vertices of degree 2 and 3 can differ by at most one and together with s_i we get for the numbers

n_2, n_3 of vertices of degree 2, resp. 3 in the boundary of P' that $n_2 - n_3 \leq 2$. With Lemma 3.1 we get

$$2 \geq h + 6 - p \Rightarrow p \geq 4$$

in contradiction to our assumption.

□

The construction of structures can often best be described by the inverse operation – the reduction. We will now describe how an mpc-patch P with at most 3 pentagons can be uniquely reduced to two smaller mpc-patches.

Let P be an mpc-patch P with at most 3 pentagons and let S be the unique cut path. As S has two endpoints on the boundary cycle of P, these split the cycle into two parts – B_1 and B_2. $B_1 \cup S$ and $B_2 \cup S$ form the boundaries of two patches Q_1, Q_2 that together contain all bounded faces of P. As the degrees in the interior of the cut path are alternating and as the endpoints have degree 2 in the boundary, Q_1 and Q_2 are pseudo-convex.

If Q_1, Q_2 have a unique vertex that can be marked, it is obvious how they can be transformed into mpc-patches. If the cyclic degree sequence of the boundary is symmetric, there is more than one vertex giving a lexicographically maximal sequence. As the marked vertex v of P is in the boundary of both – Q_1 and Q_2 – we can take v as a reference point and choose the marked vertices of Q_1 and Q_2 as the first vertices in counter clockwise direction from v that give a lexicographically maximal sequence. An example of this reduction principle can be found in Figure 3.

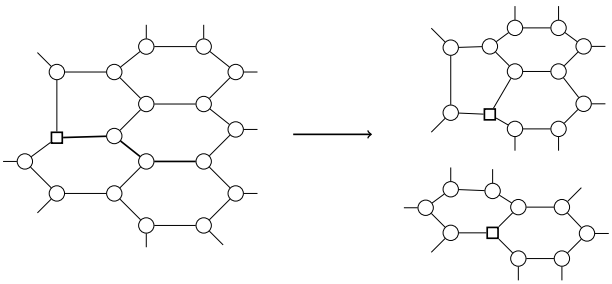


Figure 3. The reduction of an mpc-patch.

Applying this recursively to reduced parts with more than one face, we get smaller and smaller pieces and finally get a unique decomposition of each mpc-patch into faces. Reversing this reduction means that we start with faces and build increasingly larger patches.

Since cut paths go alternately right and left and start after a pair of degree 2 vertices, the border code of the reduced patch on the right hand side of the cut path has a subsequence $(0, 0, x)$ for some x. If the entry y after x is 0, the cut path is part of a subsequence $(0, x)$ or $(x + 1)$ in the left hand patch, if $y > 0$, the cut path is part of a subsequence $(0, x)$ in the left hand patch. For the construction this means that only a small fraction of all patches can serve as right hand patches.

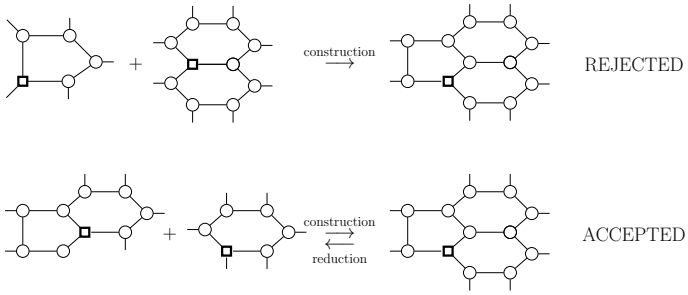


Figure 4. Accepting or rejecting an mpc-patch

We will define a construction operation that glues two mpc-patches together so that every mpc-patch with reduction (P_1, P_2) is in fact obtained by performing a construction operation with P_1 and P_2. Moreover we will only accept a newly constructed mpc-patch if it was obtained by a construction operation from the unique mpc-patches that would be the result of its reduction. This way we will only accept every mpc-patch once and keep the number of mpc-patches that have to be remembered by our algorithm relatively small. An example of this accepting/rejecting principle can be found in Figure 4.

Algorithm 1 describes the basic procedure to generate all non-isomorphic mpc-patches. To generate all non-isomorphic mpc-patches with no more than $p \leq 3$ pentagons, s hexagons and h heptagons, only newly created mpc-patches are accepted that meet the restrictions on the number of pentagons, hexagons and heptagons. Glue paths are paths in the boundary of mpc-patches along which mpc-patches can be identified. They can be

described as substrings of the border code. Our knowledge about glue paths implies that every glue path of the form $(0, x)$ in the right mpc-patch can be glued to a glue path of the form $(0, x)$ in the left mpc-patch and every glue path of the form $(0, x, 0)$ in the right mpc-patch can be glued to a glue path of the form $(x + 1)$ in the left mpc-patch.

Algorithm 1 Bottom up algorithm

```
 1: patches = {pentagon, hexagon, heptagon}
 2: current_patches = ∅
 3: new_patches = {pentagon, hexagon, heptagon}
 4: while new_patches ≠ ∅ do
 5:     current_patches = new_patches
 6:     new_patches = ∅
 7:     for each mpc-patch P in current_patches do
 8:         find glue paths in P
 9:         for each mpc-patch Q in patches do
10:             for each pair of corresponding glue paths in P and Q do
11:                 identify P and Q along the glue path to form R
12:                 put the mark of R on the first vertex of the glue path
13:                 if the reduction of R gives P and Q then
14:                     if R fulfils the restrictions then
15:                         add R to patches
16:                         add R to new_patches
17:                     end if
18:                 end if
19:             end for
20:         end for
21:     end for
22: end while
```

The number of mpc-patches increases very fast, so that the size of the set *patches* not only causes problems due to the memory consumption but especially the for-loop in line 9 of Algorithm 1 gets very time consuming. In order to reduce the number of patches that have to be stored, we use the following lemma:

Lemma 3.3. *If an mpc-patch P belongs to the set* current_patches *during the i-th time the while loop in line* 4 *of Algorithm 1 is executed (we call that the i-th iteration), then P has at least i internal faces.*

Proof. In line 3 and 5 it is assured that the statement of the lemma holds for the first iteration, so assume that it holds for iterations $1, \ldots, i-1$. The set *current_patches* in iteration i is the set *new_patches* at the end of iteration $i-1$, so it contains patches in which patches with $i-1$ internal faces are contained, but not the whole patch. So these patches have at least i internal faces. □

Let $n_{\max}$ denote the upper bound on the number of pentagons, hexagons and heptagons together and assume that in iteration i we construct an mpc-patch R (line 11) with n interior faces.

If we combined R with a patch constructed in a later iteration $j > i$, the result would have at least $n + j > n + i$ faces. If $n + i \geq n_{\max}$, such a combination is not possible, so it is not necessary to store the patch in the set *patches* (line 15). As all patches with which it – and all its descendants – can be combined are already in the list, these combinations can be constructed recursively without storing the exact structure of the patches. Even the temporary storage in line 16 can be avoided and only the information needed later for bounding the top down search must be stored (see section 4). This step can be seen as switching from a breadth first approach to a depth first approach.

stored patches (4,2);(4,2)				
f_5	f_6	f_7	A	B
0	50	0	240	532
1	10	1	213	2027
1	20	1	3085	22737
1	30	1	15632	97722
2	10	2	3798	100971

stored patches (6,1);(4,2);(4,2)				
f_5	f_6	f_7	A	B
0	4	6	231	19094
0	5	6	446	82762
0	6	6	1266	285786
0	7	6	3093	844642
1	3	7	1215	238869
1	4	7	3506	1226207

Table 1. Numbers of stored patches with optimisations (A) and without optimisations (B).

4 Top down part

While in the bottom up part the algorithm starts with the faces and builds mpc-patches, in the top down part we start with only the connecting faces and refine this structure by adding paths. These paths form successively smaller *holes* that will finally be single faces. When the holes must be filled with mpc-patches with at most 3 pentagons, the results of the bottom up part can often be used as bounding criteria – that is: to decide whether these holes can be filled or not.

Definition 6. A special face in a plane graph is a face that contains semi-edges.

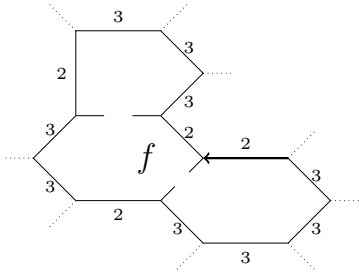


Figure 5. The canonical edge in a special face f.

Let $f = [v_0, v_1], [v_1, v_2], \ldots, [v_k, v_0]$ be a face in a plane graph. For $1 \leq i \leq k+1$ we define the *local degree* of the edge $e_i = [v_{i-1}, v_{i \bmod (k+1)}]$ with respect to the face f, denoted as $\deg_f(e_i)$, as 2 if there is no edge (in clockwise order) between $[v_{i \bmod (k+1)}, v_{i+1}]$ and $[v_{i \bmod (k+1)}, v_{(i-1) \bmod (k+1)}]$ and 3 otherwise. See Figure 5 for an example.

The local degree sequence is defined accordingly. A directed edge e in a face f is called *canonical* (for f), if the local degree sequence of f starting at e is lexicographically minimal, so unless all edges have local degree 3, a canonical edge always ends at a vertex with a semi-edge in f. You can see an example of a special face with its canonical edge in Figure 5. The dotted edges can be real or semi-edges. The numbers correspond to the local degrees of the edges in the inner face.

A pre-join is a plane connected graph where all vertices have full degree 3 and all faces are pentagons, hexagons, heptagons or special faces. Special faces f are either marked as connecting faces or carry pairwise different integer marks on a directed edge e so that the local degree sequence starting at e is lexicographically minimal. Only special faces with a local degree sequence of the form $(3,2)^l, (2,3)^m$ can be marked as connecting faces.

Faces with an integer mark on a directed edge still have to be subdivided in order to get a nanojoin. The marks give the order in which this is done by the algorithm. A nanojoin for k nanocaps with parameters $(l_1, m_1), \ldots, (l_k, m_k)$ is a special case of a pre-join. There is a bijection from the set of cap parameters to the special faces so that the local degree sequence of the i-th special face is $(3,2)^{l_i}, (2,3)^{m_i}$.

We will now describe a way to uniquely obtain a nanojoin J starting with a marked pre-join U_0. Suppose that N is a join for k caps with parameters $(l_1, m_1), \ldots, (l_k, m_k)$.

For each such parameter set (l_i, m_i) a cycle of length $2 * (l_i + m_i)$ exists with semi-edges so that the local degree sequence of both faces can be written as $(3,2)^{l_i}(2,3)^{m_i}$.

Let (l_i, m_i) be parameters so that the cycle with local degree sequence $(3,2)^{l_i}, (2,3)^{m_i}$ has smallest possible symmetry among all given parameters. Among those with the same symmetry choose (l_i, m_i), so that the triple $(l_i + m_i, l_i, m_i)$ is lexicographically maximal. The graph U_0 is a cycle with local degree sequence $(3,2)^{l_i}, (2,3)^{m_i}$ canonically marked with the number 0 on one side and as a connecting face on the other.

Starting from U_0 we recursively generate U_{i+1} from U_i until a nanojoin is constructed. Let $e = [w, v_0]$ be the marked edge of a pre-join U_i with the highest mark n. First the mark of e is removed. Starting at v_0 a path going alternately right and left into the special face is added. The end of this path can be (a) another vertex in the special face with the semi-edge inside the face (in this case the semi-edge is replaced by the last edge of the path), (b) a vertex of the path that occurs for the second time on the path, or (c) a vertex of a new cycle with degree sequence $(3,2)^{l_j}(2,3)^{m_j}$ for an index j for which no connecting cycle was added to the pre-join so far. In case (c) the interior of the cycle will be marked as a connecting face. See Figure 6, Figure 7 and Figure 8 for examples of these operations.

These operations possibly create new special faces not marked as connecting faces. The only new faces that can have the structure of connecting faces are those added in operation (c) and marked as such. Such special faces are marked with the smallest integers not yet (or no more) present in the graph. The edge that is marked is the first canonical edge in clockwise order from a canonically chosen reference edge. We choose this reference edge as an edge of the path present in all special faces that need a mark. In case (a) and (c) this is the first edge of the path, in case (b) it is more complicated to describe: if the path is $v_0, \ldots, v_k$ and j is the first index so that $v_j = v_k$, then the reference edge is the first edge in clockwise direction from $[v_j, v_{j-1}]$. This edge is contained in both new special faces.

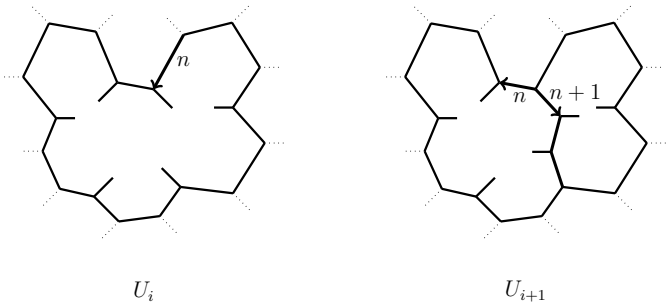


Figure 6. Extending a pre-join (a)

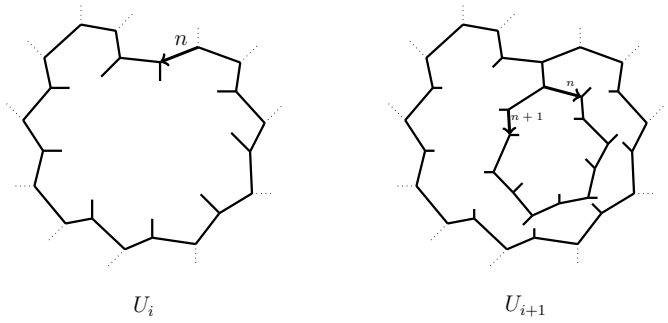


Figure 7. Extending a pre-join (b)

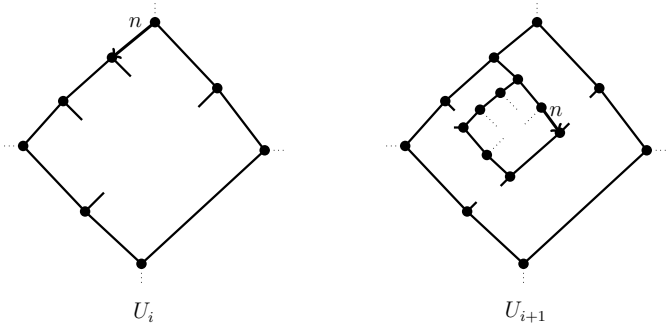


Figure 8. Extending a pre-join (c)

Decomposing a given join starting from an edge satisfying the requirements for the first marked edge in U_0, it is immediately clear that every nanojoin can be obtained

in this way. This implies that applying the recursive extension steps in each possible way consistently with the initial parameters for the connecting faces and the number of hexagons, heptagons and pentagons, each nanojoin is constructed at least once.

The decomposition is unique for a given unique U_0 as a subgraph of a join. Nevertheless isomorphic joins are constructed if the parameters (l_i, m_i) chosen for U_0 occur at least twice, we have $m_i = 0$, or we have $l_i = m_i$ – which implies that for all parameters (l_k, m_k) we have $l_k = m_k$ or $m_k = 0$. These isomorphic nanojoins will be filtered out in the final step where we use a coarser equivalence relation as our concept of isomorphism anyway.

The recursive subdivision of faces with integer marks until all faces not marked as connecting faces are pentagons, hexagons or heptagons, can be implemented as a stand-alone program that can generate nanojoins. Nevertheless it turned out that for most parameter sets it is more efficient to combine it with the bottom up approach that gives bounding criteria that allow relatively early detection of pre-joins that can't be completed to form nanojoins for the given parameters.

5 The combined approach

In the top down approach it often happens that combinations of special faces are constructed that can not be filled in with the numbers of pentagons, hexagons or heptagons given as parameters. An efficient way to predict whether a given combination of special faces can be filled in, can be obtained from our bottom up approach.

When all nanojoins with p pentagons, s hexagons, and h heptagons are to be generated, first the bottom up part is applied to generate all mpc-patches for some $p' \leq \min\{p, 3\}$, $h' \leq h$ and $s' \leq s$. These results are used to detect a lot of cases where a given special face can not be filled in. Tests showed that this is more efficient than actually storing the lists of patches that can be filled in. Instead of storing the mpc-patches themselves, we store four integers representing the minimum number of pentagons, hexagons, heptagons and internal vertices of the mpc-patch together with its border code.

All possible border codes are stored in a prefix tree. Vertices of the tree represent the border code of an mpc-patch. They contain extra information that tells us what the minimum number of pentagons, hexagons, heptagons and internal vertices for an mpc-patch with that border code is. It is important to note that those four numbers do not

necessarily belong to the same mpc-patch, so that according to the information of the tree it could be possible to fill a certain boundary with the given numbers of pentagons, hexagons, heptagons and internal vertices, but nevertheless no single patch that fulfils all requirements exists. An example of a prefix tree can be found in Figure 9.

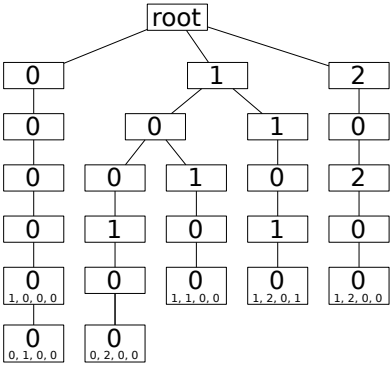


Figure 9. A prefix tree for mpc-patches with at most 1 pentagon, 2 hexagons and no heptagons

Next we start the top down algorithm. We decide on parameters $p' \leq \min\{p, 3\}$, $h' \leq h$ and $s' \leq s$ to be used for the bounding criteria given by the bottom-up algorithm. We keep track of minimum numbers of faces of every size necessary for parts that have already been filled in or for which lower bounds are known. When the number of pentagons, hexagons and heptagons left for the special face U we want to fill in is respecitively smaller than p', s' and h', we can apply the following optimisation. Whenever we split U, we assign numbers of connecting faces to be contained in the new special faces in every possible way. For those new special faces that are assigned 0 connecting faces and have to be filled with pseudo-convex patches, we check whether those special faces can be filled in and whether the sums of the lower bounds for the number of pentagons, hexagons, heptagons and internal vertices we know to be necessary, is still within the range of the given parameters.

It may look inefficient to distribute the number of connecting faces between the new special faces in every possible way and treat those as different pre-joins, as this leads to a bigger branching factor. Nevertheless tests show that this allows us to discard pre-joins faster and speeds up the algorithm.

6 Isomorphism rejection

Joins never occur as isolated molecules but always as substructures of large molecules. This must be taken into account when defining the concept of *equivalent* or *isomorphic* nanojoins. Adding e.g. an extra ring of hexagons to a connecting face in a nanojoin N results in a nanojoin N' that is obviously not isomorphic to N as a graph, but must be considered equivalent as a nanojoin as being part of a larger structure – where typically the tube body attached to the connecting faces is very large compared to the size of the join. The only difference between these nanojoins is where the boundary between the join and the tube body is chosen.

Definition 7.

- For a nanojoin N we write $\bar{N}$ for the infinite 3-regular graph obtained from N by identifying the boundary of each connecting face with the boundary of a halftube with the same parameters as the face. We call $\bar{N}$ an extended nanojoin.
- We say that two nanojoins N, N' are isomorphic as joins if $\bar{N}$ and $\bar{N}'$ are isomorphic as plane graphs.

Joins without pentagons or heptagons only exist for two connecting faces with the same parameters and up to isomorphism there is only one such join, so we will focus on the case where there are heptagons and possibly also pentagons.

For an extended nanojoin $\bar{N}$ and a directed edge $[v, w] \in \bar{N}$ we define an infinite string $s([v, w])$ as follows. First the vertices $x \in \bar{N}$ are assigned labels $l(x)$ in a breadth first manner:

(1) $l(v) = 1$, $l(w) = 2$

(2) if x is the vertex with smallest label that has still unlabelled neighbours, the neighbours of x are inspected – and labelled with the next not yet assigned label in case they are not yet labelled – in clockwise order starting at the neighbour with smallest label.

Note that as $\bar{N}$ is connected and infinite, there is always a vertex x in (2) with the required properties and it has always an already labelled neighbour.

The string $s([v, w])$ is then defined as the infinite string

$a_1, a_2, a_3, \ldots$ with $a_{3(i-1)+1}, a_{3(i-1)+2}, a_{3(i-1)+3}$ the labels of the three neighbours of the vertex labelled i in clockwise direction starting with the smallest label.

It is easy to see that if for two oriented edges $[v,w] \in \bar{N}$, $[v',w'] \in \bar{N}'$ we have $s([v,w]) = s([v',w'])$, then there is an isomorphism (or in case of $\bar{N} = \bar{N}'$ an automorphism) from $\bar{N}$ to $\bar{N}'$ mapping $[v,w]$ to $[v',w']$ – and that if there is an isomorphism mapping $[v,w]$ to $[v',w']$, then $s([v,w]) = s([v',w'])$.

Definition 8. Let $\bar{N}$ be an extended nanojoin with a finite positive number of heptagons. Let $\bar{N}_i$ be the mirror image of $\bar{N}$.

If there is at least one pentagon, $E_0(\bar{N})$ denotes the set of all oriented edges with a pentagon on the left – otherwise $E_0(\bar{N})$ denotes the set of all oriented edges with a heptagon on the left.

Define the mirror parameters of a connecting face with parameters (l,m) as (m,l) if $m \neq 0$ and $(l,0)$ if $m = 0$. If the set of mirror images of all parameters of $\bar{N}$ is different from the set of parameters, then we define $E_1(\bar{N}) = E_0(\bar{N})$, otherwise we define $E_1(\bar{N}) = E_0(\bar{N}) \cup E_0(\bar{N}_i)$

The canonical string of $c(\bar{N})$ is defined as

$c(\bar{N}) = \min\{s([v,w])|[v,w] \in E_1(\bar{N})\}$

with the strings ordered according to the lexicographic ordering of strings.

Mapping vertices onto each other that get the same labels in a canonical labelling, one can easily see that two extended nanojoins are isomorphic if and only if their canonical codes are the same.

In order to judge whether two extended nanojoins are isomorphic, we have to decide whether their canonical codes are the same. We will show that it is sufficient to compare some finite prefix of the codes.

Theorem 6.1. *Let N_1, N_2 be two joins with the same number of pentagons and heptagons.*

If there is an isomorphism $\phi()$ from N_1 onto a subgraph S of $\bar{N}_2$ then $\bar{N}_1$ and $\bar{N}_2$ are isomorphic.

Proof. The complement S^c of S in $\bar{N}_2$ contains only hexagons, as all pentagons and heptagons in N_1 are mapped onto pentagons and heptagons in S. Let C be a connecting face in N_1. If a hexagon in $\bar{N}_1 - N_1$ or a pair of hexagons in $\bar{N}_1 - N_1$ sharing an edge would have a disconnected intersection with C, at least one of these hexagons would contain two edges of C. In each case there would be a path P connecting two vertices of degree 2 in C

that contains no, one or two vertices of $\bar{N}_1 - N_1$ and in case of two internal vertices turns once right and once left. $C \cup P$ would bound two regions with faces in the complement of N_1 and one of them would have to be a patch consisting only of hexagons. From the structure of P and C it follows that with n_i the number of vertices of degree i in the boundary of the patch we would have $n_2 - n_3 \leq 5$. Lemma 3.1 would then imply that the patch would contain at least one pentagon. This is a contradiction, so no face or pair of faces sharing an edge in $\bar{N}_1 - N_1$ can have a disconnected intersection with C.

This implies that the subgraph $B(C)$ induced by the vertices in $\bar{N}_1 - N_1$ that share a face with vertices in C is a simple cycle with the same degree sequence and the same is true for the subgraph induced by the vertices in $\bar{N}_2 - S$ that share a face with vertices in $\phi(C)$. This allows to uniquely extend the isomorphism to the nanojoin $N_1 \cup B(C)$ and by induction to all of $\bar{N}_1$. □

Definition 9. Let $s = s_1, s_2, s_3 \ldots$ be an infinite string. We say that a string $t_1, t_2, \ldots, t_k$ is a prefix of s if $s_i = t_i$ for $1 \leq i \leq k$.

If for a nanojoin N the string $c_0(\bar{N})$ is a prefix of $c(\bar{N})$ that contains for all vertices v in N the list of labels of the neighbours of v, then we call $c_0(\bar{N})$ a defining prefix of $c(\bar{N})$.

Corollary 6.2. *Let N_1, N_2 be two joins with the same number of pentagons and heptagons and $c_0(\bar{N}_1)$ a defining prefix of $c(\bar{N}_1)$.*

If $c_0(\bar{N}_1)$ is also a prefix of $c(\bar{N}_2)$, then N_1 and N_2 are isomorphic and also $c(\bar{N}_1) = c(\bar{N}_2)$.

Proof. Mapping the vertices of N_1 onto the vertices of N_2 that get the same label in the canonical string, we have an isomorphism of N_1 onto a subgraph of N_2, so that Theorem 6.1 gives the required result. □

Corollary 6.2 is used to detect whether a newly found nanojoin is isomorphic to one that was previously found by the algorithm. When we find a new join N, we first attach rings of hexagons to every connecting face so that every vertex that is part of the defining prefix of $c(\bar{N})$ is part of the join with the rings of hexagons attached. This way we can compute the defining prefix $c_0(\bar{N})$.

At each moment we have the defining prefix of one representative of each isomorphism class of joins for which we already constructed an element in a prefix tree. Whenever we have a new defining prefix c_0, we check whether there is a match in the tree – that is:

whether a prefix of c_0 is already in that tree or whether c_0 is a prefix of a code in the tree. If there is no match, the join is output and c_0 is added to the tree, otherwise we discard c_0 and continue the construction. This test can be done in time that is linear in the size of c_0.

This method of guaranteeing that only non-isomorphic joins are generated is very straightforward. In most generation programs methods are used that guarantee the generation of only non-isomorphic structures without actually comparing structures (see [1] for an overview of these methods). Such methods are necessary in cases where the construction of the structures is very fast and easy and isomorphism rejection is the bottleneck. In this case the construction of the nanojoins is the crucial part and bottleneck and the cost for using the lists was less than 2% of the generation time in all cases we tested. Furthermore the lists are never so large that keeping them in memory is any problem, so that neither time consumption nor memory consumption make the use of more sophisticated methods necessary.

7 Testing and Results

This section will discuss some results on the number of nanojoins with up to four caps. We will also give the running times the program needed to generate those nanojoins. Our program was compiled using gcc and executed without parallelisation on an Intel Xeon E5-2690 CPU at 2.90GHz on a machine having 252GB memory. The joins that are found by the program are written to a file in planar code format (for a definition of this format see the homepage of *CaGe* [6]).

Since we already had an extended version of the bottom up algorithm that was able to generate the nanojoins completely independent from the top down algorithm and the top down algorithm can be run without the bottom up part, we have two completely independent algorithms. This can be used to test the correctness of our algorithm - e.g. to check whether both algorithms find the same joins. We also checked those results against the combined approach to make sure that that approach was still working correctly.

In total we tested 5 different versions of our generator. The first two are the standalone bottom up algorithm and the top down algorithm. The other three all used the combined approach, but with different values for p', h' and s' determining when the bottom up part is used:

- The full combined approach ($p' = min(p, 3)$, $h' = h$ and $s' = s$)
- The no-pentagon combined approach ($p' = 0$, $h' = h$ and $s' = s$)
- The mixed combined approach that uses the full combined approach if $h \leq 6$ and otherwise the no-pentagon approach

The mixed combined approach was chosen because in some cases the no-pentagon combined approach performed better while in other case the full combined approach did. The mixed combined approach is chosen in a way to use the full combined approach in the cases where it normally performs better and the no-pentagon combined approach in the cases where it normally doesn't. All results presented are obtained by the mixed combined approach.

It is important to note that the amount of memory used by the combined algorithm directly depends on the values for p', h' and s' and can get very high very soon. When the amount of memory is scarce it is necessary to decrease these internal values of the program.

The following tables contain some results. We ran our algorithm to generate joins going from two to four caps and listed the running times for the combined, bottom up and top down approach together with the number of joins found by our algorithm. They were confirmed to be the same by all approaches.

The tables show that the bottom up algorithm generally performs well when the number of pentagons and heptagons is small, as this implies a relatively small number of patches. This also implies that the bottom up algorithm is generally better for two caps and the top down algorithm for three caps.

For four caps, the bottom up algorithm always performs better. This is because the number of hexagons is still small for our test cases. The small number of hexagons again leads to a significant decrease in possible patches.

That is why we chose the combined approach that makes sure the number of patches generated in its bottom up part does not get too big. The tables show that the combined approach performs much better than the other approaches in a lot of cases.

cap parameters	p	h	s	joins	combined	bottom up	top down
(6-0) (6-0)	0	35	0	1	38ms	1ms	133s
(4-2) (3-2)	1	21	1	1	0.3s	0.3s	672s
(6-0) (5-2)	1	23	1	1	0.1s	0.5s	632s
(6-0) (6-0)	1	23	1	1	0.1s	49ms	648s
(4-2) (3-2)	2	16	2	194	2.6s	12s	746s
(6-0) (5-2)	2	18	2	174	2.6s	20s	770s
(6-0) (6-0)	2	18	2	69	2.8s	2.2s	900s
(4-2) (3-2)	3	12	3	5491	16s	116s	556s
(6-0) (5-2)	3	14	3	4279	30s	267s	725s
(6-0) (6-0)	3	14	3	464	30s	30s	903s
(4-2) (3-2)	4	9	4	40569	74s	493s	581s
(6-0) (5-2)	4	10	4	19808	62s	825s	306s
(6-0) (6-0)	4	11	4	2570	120s	195s	1086s

Table 2. Results for joins with two caps

cap parameters	p	h	s	joins	combined	bottom up	top down
(4-2) (3-2) (5-1)	0	10	6	631	39s	291s	609s
(6-0) (5-0) (4-1)	0	12	6	172	141s	810s	321s
(6-0) (5-2) (4-2)	0	12	6	522	143s	1043s	978s
(6-0) (6-0) (6-0)	0	13	6	12	265s	233s	586s
(6-0) (5-2) (4-2)	1	7	7	1650	38s	1038s	104s
(6-0) (6-0) (6-0)	1	9	7	39	120s	952s	267s
(6-0) (5-2) (4-2)	2	4	8	1721	18s	647s	58s
(6-0) (6-0) (6-0)	2	5	8	20	13s	399s	57s
(6-0) (5-0) (4-0)	0	12	6	61	141s	129s	769s
(6-0) (5-0) (5-0)	0	13	6	37	265s	225s	882s
(6-0) (6-0) (5-1)	0	12	6	54	140s	800s	106s
(4-2) (3-2) (5-1)	1	6	7	3768	68s	428s	182s
(6-0) (6-0) (5-1)	1	7	7	167	18s	835s	16s
(4-2) (3-2) (5-1)	2	4	8	10303	154s	815s	367s
(6-0) (6-0) (5-1)	2	4	8	147	3.7s	477s	18s

Table 3. Results for joins with three caps

cap parameters	p	h	s	joins	combined	bottom up	top down
(6-0) (5-0) (4-0) (3-0)	0	1	12	1	2.6s	1.6s	307s
(6-0) (6-0) (6-0) (6-0)	0	4	12	4	571s	748s	928s
(6-1) (4-2) (5-1) (3-3)	0	1	12	5	3.6s	6s	460s
(6-0) (5-0) (4-0) (3-0)	1	1	13	2	481s	46s	18939s
(6-0) (6-0) (6-0) (6-0)	1	1	13	1	3.9s	38s	307s

Table 4. Results for joins with four caps

We also looked at the number of joins found by our combined algorithm for the maximum number of hexagons (but at most 60) while keeping the running time under 20 minutes. These results are presented in Table 5.

Table 5. Results obtained by combined approach

cap parameters	p	h	s	**joins**[1]	**joins**[2]	**time**
(4-2) (3-2)	0	60	0	0	0	3.3s
(6-0) (5-2)	0	60	0	0	0	67ms
(4-2) (3-2)	1	60	1	1678	1	85s
(6-0) (5-2)	1	60	1	271	1	19s
(6-0) (6-0)	1	60	1	10	1	19s
(4-2) (3-2)	2	40	2	132850	867	1086s
(6-0) (5-2)	2	47	2	42969	1176	1054s
(6-0) (6-0)	2	47	2	2596	561	1188s
(4-2) (3-2)	3	21	3	586074	33855	883s
(6-0) (5-2)	3	23	3	153361	30662	903s
(6-0) (6-0)	3	23	3	6754	4172	891s
(4-2) (3-2)	4	12	4	612236	146448	646s
(6-0) (5-2)	4	14	4	243671	119726	781s
(6-0) (6-0)	4	14	4	11309	10463	751s
(4-2) (3-2) (5-1)	0	15	6	61013	2972	940s
(6-0) (5-0) (4-1)	0	15	6	2019	565	866s
(6-0) (5-2) (4-2)	0	15	6	7069	1365	934s
(6-0) (6-0) (6-0)	0	15	6	27	20	868s
(6-0) (5-2) (4-2)	1	10	7	29004	13247	820s
(6-0) (6-0) (6-0)	1	11	7	142	138	669s
(6-0) (5-2) (4-2)	2	7	8	57211	39659	720s
(6-0) (6-0) (6-0)	2	9	8	901	891	1115s
(6-0) (5-0) (4-0)	0	15	6	278	191	831s
(6-0) (5-0) (5-0)	0	15	6	115	83	866s
(6-0) (6-0) (5-1)	0	15	6	619	177	866s
(4-2) (3-2) (5-1)	1	8	7	57514	15023	627s
(6-0) (6-0) (5-1)	1	11	7	5114	2383	684s
(4-2) (3-2) (5-1)	2	5	8	58686	31249	533s
(6-0) (6-0) (5-1)	2	9	8	32486	18282	1166s
(6-0) (5-0) (4-0) (3-0)	0	4	12	43	42	608s
(6-0) (6-0) (6-0) (6-0)	0	4	12	4	4	571s
(6-1) (4-2) (5-1) (3-3)	0	4	12	3646	2181	1016s
(6-0) (6-0) (6-0) (6-0)	1	3	13	1	1	446s

8 Future work

The result of the generation is the combinatorial structure of the nanojoins as plane graphs. In order to produce realistic 3D embeddings and add the generator to the environ-

ment CaGe (see [6]) it is necessary to develop specialized programs for fast 3D-embedding of nanojoins. This will be done in the near future.

References

[1] G. Brinkmann, Isomorphism rejection in structure generation programs, in: P. Hansen, P. W. Fowler, M. Zheng (Eds.), *Discrete Mathematical Chemistry*, Am. Math. Soc., Providence, 2000, pp. 25–38.

[2] G. Brinkmann, J. Goedgebeur, B. D. McKay, The generation of fullerenes, *J. Chem. Inf. Model.* **52** (2012) 2910–2918.

[3] G. Brinkmann, D. Mourisse, L. Rylands, On the existence of nanojoins with given parameters, *J. Math. Chem.* **53** (2015) 2078–2094.

[4] G. Brinkmann, U. von Nathusius, A. H. R. Palser, A constructive enumeration of nanotube caps, *Discr. Appl. Math.* **116** (2002) 55–71.

[5] M. S. Dresselhaus, G. Dresselhaus, P. C. Eklund, *Science of Fullerenes and Carbon Nanotubes*, Academic Press, San Diego, 1996.

[6] G. Brinkmann, O. Delgado–Friedrichs, S. Lisken, A. Peeters, N. Van Cleemput, Cage – a virtual environment for studying some special classes of plane graphs – an update, *MATCH Commun. Math. Comput. Chem.* **63** (2010) 533–552. `http://caagt.ugent.be/CaGe/`.

[7] G. W. Ho, A. T. S. Wee, J. Lin, Electric field–induced carbon nanotube junction formation, *Appl. Phys. Lett.* **79** (2001) 260–262.

[8] E. Lijnen, A. Ceulemans, M. V. Diudea, C. L. Nagy, Double toroids as model systems for carbon nanotube junctions: through–bond currents, *J. Math. Chem.* **45** (2009) 417–430.

[9] D. Mourisse, Generatie van nanojoins, Master's thesis, Universiteit Gent, 2013.

[10] R. F. Service, Mixing nanotube structures to make a tiny switch, *Science* **271** (1996) 1232–1232.

[11] K. Tserpes, I. Konstantinos, P. Papanikos, Continuum modeling of carbon nanotube–based super–structures, *Composite Struct.* **91** (2009) 131–137.

[12] I. Zsoldos, Planar trivalent polygonal networks constructed from carbon nanotube y-junctions, *J. Geom. Phys.* **61** (2011) 37–45.