Available online at www.sciencedirect.com
SCIENCE DIRECT®
ELSEVIER
Discrete Applied Mathematics 128 (2003) 541–554
DISCRETE APPLIED MATHEMATICS
www.elsevier.com/locate/dam

The construction of cubic and quartic planar maps with prescribed face degrees

Gunnar Brinkmann[a], Thomas Harmuth[b], Oliver Heidemeier[c]

[a] *Fakultät für Mathematik, Universität Bielefeld, Postfach 10 01 31, D-33501 Bielefeld, Germany*
[b] *Am Buschkotten 9, D 33739 Bielefeld, Germany*
[c] *Huelshorstweg 30, D 33415 Verl, Germany*

Received 15 February 1999; received in revised form 13 May 2002; accepted 8 July 2002

Abstract

In this paper, the existence and availability of computer programs to constructively enumerate all simple connected cubic or quartic planar maps with prescribed number of vertices and face degrees is announced and results of the programs are presented. The underlying algorithms of the computer programs are described.

Keywords: Regular graph; Embedding; Face degree; Structure generation

1. Introduction

The degree $deg(f)$ of a face f in a connected planar map $G=(V,E)$ is given by $deg(f) := |\{e \in E \mid e \text{ is in the boundary of } f \text{ and } e \text{ is not a bridge}\}| + 2 \cdot |\{e \in E \mid e \text{ is a bridge in the boundary of } f\}|$-that is the degree of the corresponding vertex in the dual graph.

A lot of problems in mathematics and chemistry (see e.g. [1,4,10,11,15,16] deal with regular maps where only certain face degrees may occur, that is: there is a set $S=\{f_1,\ldots,f_k\}$, $f_1,\ldots,f_k \in \mathbb{N}$ and all face degrees are required to be in S. E.g. the well-known fullerenes (see [14]) can be defined as cubic planar maps where all faces are pentagons or hexagons ($S=\{5,6\}$). Let us call planar maps with all face sizes

E-mail addresses: gunnar@mathematik.uni-bielefeld.de (G. Brinkmann), thomas@harmuths.de (T. Harmuth), oliver.heidemeier@mediaways.net(O. Heidemeier).

0166-218X/03/$ - see front matter
PII: S0166-218X(02)00549-8

in $S \subset \mathbb{N}$ and regular of degree $d(d,S)$-maps. In order to check conjectures for given d and S up to maps of a certain size or determine the energetically best molecule corresponding to a (d,S)-map with a given number of vertices (atoms), it is useful to have complete lists of all non-isomorphic (d,S)-maps for the number of vertices in question available.

The aim of this paper is to announce the availability of computer programs for this purpose and present the main ideas of the underlying algorithms, that for given $S \subset \mathbb{N}$, $d \in \{3,4\}$ and $n \in \mathbb{N}$ can list all non-isomorphic planar (d,S)-maps with n vertices. The only algorithm previously applied for this task is the spiral algorithm originally proposed for fullerenes in [16], but used also for other classes of cubic graphs (see e.g. [1,8]). The problem of this algorithm is on one hand that it is fairly slow and—an even more serious problem—on the other that it is known to miss some structures (see e.g. [2,9]). Except for the algorithm presented in [5], which is specialized on fullerenes, the algorithms presented in this paper are the only algorithms for this task that can guarantee the generation of complete lists and—implemented as a computer program—are fast enough to construct lists of interesting size. In this article we can only present the main ideas of the algorithms. A detailed treatment can be found in [12,13].

If $F(G)$ denotes the set of faces of the map G, as a consequence of the Euler formula in cubic planar maps we have $\sum_{f \in F}(6 - deg(f)) = 12$ and in quartic planar maps we have $\sum_{f \in F}(4 - deg(f)) = 8$. So quartic maps always contain triangles while cubic maps always contain triangles, squares and/or pentagons.

The algorithms presented here are based on the one described in [5]. The construction method for fullerenes described in there could be directly applied for cubic maps with maximum face degree 6. For larger faces and the quartic case some essential changes were necessary.

2. Part 1: The algorithm

The maps will be constructed by gluing together smaller parts. The construction can best be described by the inverse process, that is: cutting the map into the parts it was assembled of.

The main strategy is to cut the map into parts along some well-defined *cut paths*. In the cubic case we use *Petrie paths* (see [7]), that is: paths that whenever approaching a new vertex take the next edge in clockwise (left) and counterclockwise (right) direction around this vertex interchangingly. In the quartic case we use *straight paths*, that is paths that when approaching a new vertex continue at the second edge in clockwise (or equivalently counterclockwise) orientation around the vertex.

For a given directed edge e_0 and first direction r (left or right in the cubic case, straight in the quartic case), this way it is uniquely defined what will be the next edge e_1, adjacent to e_0 in the cut path. Since the graph is finite, enlarging the path this way step by step there is some n such that the endpoint of e_n is already contained in the path. If this vertex is the starting vertex of e_0 and the next edge and direction to be chosen are e_0, r, we have a closed *Jordan curve path* (see Fig. 1) —otherwise we stop

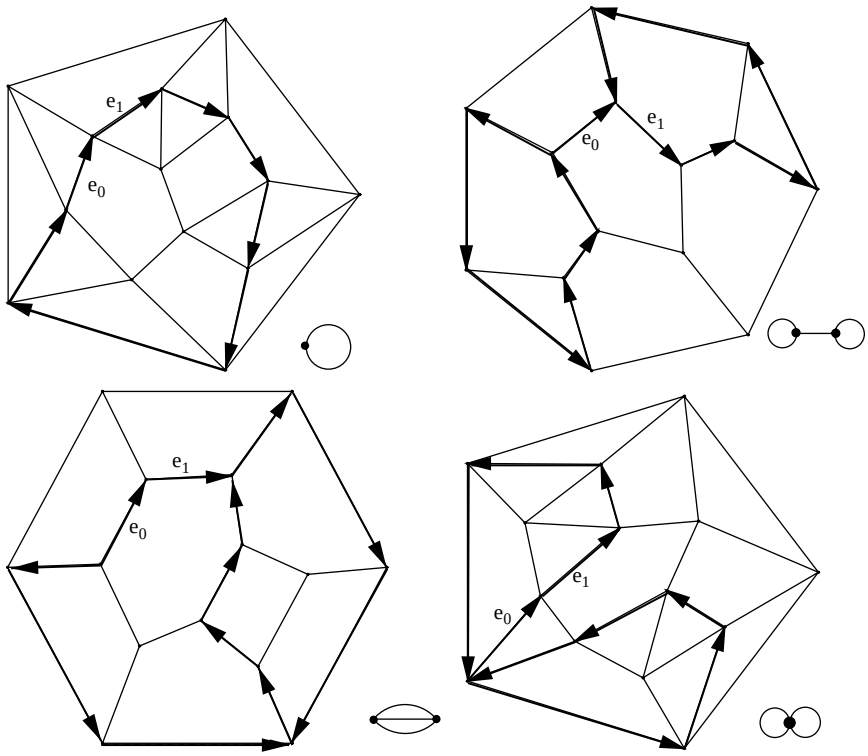

Fig. 1. A Jordan curve path, a dumb-bell path, a sandwich path and a figure 8 path.

and reverse the direction, that is: take the inverse of e_0 and direction r and proceed to construct the path, using edges $e_{-1}, e_{-2}, \ldots$, until the end vertex of e_{-m} is already contained in the path $e_n, e_{n-1}, \ldots, e_0, \ldots, e_{-(m-1)}$. In this case we stop. The two points (which may coincide in the quartic case) where the path meets previously visited points have a minimum valency of 3 in the subgraph formed by the edges visited, while the others have valency 2. Thus, in the cubic case there are two fundamentally different possibilities for such paths: once there may be 3 disjoint paths between these two vertices (this is called a *sandwich path*) and on the other hand there may be one path between the vertices and one loop path at every vertex (this is called a *dumb-bell path*). In addition to these cases in the quartic case, the two vertices of a dumb-bell path may coincide, giving one vertex of degree 4. In this case we get a *figure* 8 *path*. See Fig. 1 for examples of these path types.

Definition 1. A 3-*patch* (resp. 4-*patch*) is a planar map with all the boundary vertices of valency 2 or 3 (resp. 2,3 or 4) and all the interior vertices of valency 3 (resp. 4).

A 4-patch is called pseudo-convex if all boundary vertices are of degree 2 or 3, while a 3-patch is called pseudo-convex if there are no two vertices sharing a boundary edge that both have degree 3.

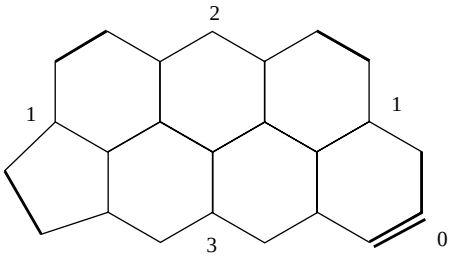

Fig. 2. A patch with a canonically marked 0-edge.

It can be easily seen that the two or three parts cut out of cubic or quartic maps by the corresponding cut paths are all pseudo-convex 3- resp. 4-patches.

If a patch has a certain symmetry of the boundary that is not a symmetry of the interior part—like it is in general the case especially for the patches cut out by a Jordan curve path—the various possibilities to insert the patch into the path will mostly result in different maps. So in order to describe a map uniquely, it is not only necessary to give a cut path and the patches cut out by the cut path, but it is also necessary to describe uniquely how the patches have to be glued into the path.

To this end we put marks on the patches and each path segment bounding a region and require that the patches are glued to the path in a way that the marks match. In the cubic case we mark edges, while in the quartic case we mark vertices.

Since we can choose where to put the mark—we just have to mark the path and the patches correspondingly—we can require the following (described from the viewpoint of the patches): If a vertex of valency 2 exists in a 4-patch, then the mark must be carried by such a vertex (we call it *markable* vertex) and if an edge with both endpoints of valency 2 exists in the cubic case, then the mark must be carried by such a *markable* edge. Otherwise, all vertices (edges) are markable.

In case we have markable vertices in the quartic case, we can compute the number of edges between any two markable vertices and obtain a cyclic sequence by listing the numbers in clockwise orientation around the boundary. We choose the mark to be put on a markable vertex, so that the sequence started at this vertex is lexicographically maximal. This sequence is called the *boundary sequence* of the patch. Of course in the cubic case we can work analogously by counting the number of vertices of valency 3 between two markable edges.

Patches that are marked this way are called *canonically marked* patches. See Fig. 2 for an example of a canonically marked patch.

So, if we want to generate all patches that can be glued to a canonically marked cut path, we have to construct all canonically marked patches. We regard marked patches as isomorphic if and only if there is an orientation preserving isomorphism of the patches mapping the marks onto each other.

If we had all the canonically marked pseudo-convex patches that might occur in the maps which we want to generate (described by the number of vertices and the face

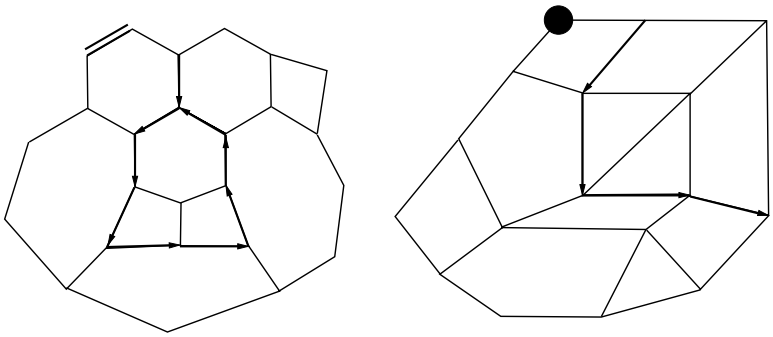

Fig. 3. Two canonically marked patches and the way they are cut.

degrees that may occur), then by enumerating all possible cut paths, putting a mark to the corresponding places and inserting all combinations of patches, always gluing mark to mark, it would be possible to construct all maps.

3. Part 2: generating the list of marked patches

In order to describe how we construct the set of all canonically marked pseudo-convex patches, we will again describe the inverse process, that is how larger marked patches can be reduced to smaller ones. We use the same strategy already used to cut the complete map into parts: we use a cut path to split the pseudo-convex patch into two smaller ones.

We apply a similar strategy already applied for cutting the map: starting at the first vertex of degree 3 in clockwise direction of the mark, we cut the patch along a cut path until we reach the boundary of the patch or the cut path approaches itself. See Fig. 3 for examples of these two possibilities. In the cubic case the cut path chosen always starts by going to the right at the first inner vertex.

It can be easily seen that the two resulting patches are again pseudo convex. By defining that the mark on the two smaller pseudo-convex patches must be put on the first canonical position in counterclockwise direction from the vertex where the cut path started, we have uniquely defined two ancestors for the marked path, so the reverse of the cutting operations give a recursive construction of the set of canonically marked pseudo-convex patches:

As patches with only one face we take one face of every degree that is allowed to occur in the map, marked at an arbitrary edge resp. vertex. Then we proceed by constructing and storing the patches with increasing number of faces by always combining smaller patches in every possible way: we take every smaller patch once as the first and once as the second patch in the pair and glue them together along boundary segments in every way that gives a pseudo-convex patch. The mark is put on the first vertex (edge) of the patch coming from the first patch in counterclockwise direction of the cut path.

During the construction we must check whether the patch obtained would be reduced in the same way we just constructed it—that is: whether the new mark is a canonical mark, whether the old canonical marks are the first ones in counterclockwise direction from the cut and whether we did not obtain any double edges. Due to the fact that vertices are identified, in the cubic as well as in the quartic case, we sometimes do get double edges. If one of these conditions does not hold, the patch is rejected.

4. Part 3: storing the patches

Of course it is essential that the patches are stored in a way that they are easily accessible on one hand and do not use too much memory on the other. We have chosen to store the patches in a tree-like structure. Branching according to the number of faces at the root and according to the entries in the boundary sequence later on, a patch that might fit into a cut path can easily be found or be detected not to exist.

For every patch P we store the following information: if P has no ancestors, then P consists of only one face. We store the degree of this face. If P has ancestors A and B, then we store two pointers to these ancestors and information on how they have to be assembled with respect to the marks of A and B (e.g. the distances between the beginning of the cut path and the marks of the smaller patches which already determine the length of the cut path). This information is sufficient to reconstruct P from A and B and needs only a constant small amount of memory—independent of the size of the patch.

5. Part 4: constructing the maps

We construct every possible cut path that might be contained in the maps we are looking for and put marks on it in a well defined way, so that the hypothetical fillins would be canonically marked.

If we e.g. have a Jordan curve path in the quartic case, it is well defined by its length. We can put the mark on the first vertex of the closed path and say that the first patch is on the right and the second on the left. If we have e.g. a dumb-bell path in the cubic case, we can call the inverse of the edge with smallest index ($e_{-(m-1)}$) the first edge. In this case the path is well defined by the lengths of the 3 segments between the points of valency 3 and the requirement to turn right at $e_{-(m-1)}$. This additional requirement is justified, since for every cubic map with a dumb-bell path in at least one of the map and its mirror image there is a dumb-bell path that turns right at the first edge and we just want to generate one of them. The marks are put on the first canonical positions seen along the path, the first patch to be filled is the patch bounded by the loop containing the starting edge, the second is the non-loop and the third is the other loop. Though the details are a bit different in the remaining cases (e.g. sandwich path cubic, dumb-bell path quartic, etc.), the general strategy should be clear.

Then we check our list of patches, whether for the boundary sequence given by the path there are patches that can be glued in (gluing the mark of the patch to the mark in the path). If this is possible, the various patches are combined in every possible way. In fact, by going through the various boundary sequences in the list and only constructing those paths where at least the boundary sequence just considered matches one of the parts, it can be avoided to construct paths where none of the parts can be filled.

6. Part 5: testing for isomorphisms

The method for isomorphism rejection is called *generating structures in canonical form* (see [3] for a survey).

The general principle is to establish a one to one correspondence between the way a structure is generated and a string. Among all possible strings encoding a possible construction of the map (that is: all possible ways how it can be generated) one string is chosen as the *canonical* string, in our case the lexicographically minimal one. For each newly constructed map, *all* strings corresponding to possible constructions are computed and the map is accepted if and only if the original string is canonical. Since every possible construction (up to automorphisms) of a structure is performed exactly once, this way it is made sure that every structure is accepted exactly once.

In the following, we will sketch how we applied this method.

It is easy to see that the construction can be uniquely described by a map plus a single directed edge in the map: Given the map and the first edge of the cut path used to construct it—directed from the first vertex of the path to the second—it is easy to construct the whole Petrie, resp. straight path used to construct the map. Following the rules in Part 4 to put the marks and number the regions, we can easily determine the marked patches to be glued in and the order in which they have to be glued into the regions—that is: we can reconstruct the whole generation process from this information.

Now let us describe a string that encodes the structure of the planar map with respect to some directed edge in the map and an orientation σ (clockwise or counterclockwise):

We label the starting vertex of e as 1 and the endvertex 2.

Then we label the other neighbours of 1 in the prescribed orientation starting at e with $3, 4$, etc. The *reference edge* for a vertex is the inverse of the edge along which it was first seen. Having labelled all neighbours of the vertices labelled 1 to $x < n$, we label the neighbours of the vertex v labelled $x + 1$ that are not yet labelled with the smallest number not used so far, starting at the reference edge of v and proceeding around v in the direction given by σ.

To this labelling we associate the following code: for every vertex we compute a linear list of the labels of the neighbours, starting with the endpoint of the reference edge and proceeding in the orientation σ. Each list is ended with a 0. By concatenating these lists in the order given by the labels of the vertices we get the code.

Having constructed a map, we compute the associated string given by this code starting at the first edge of the cut path and taking σ as counterclockwise

orientation. So the directed edge is always $(1,2)$ and the map is completely encoded in this string.

Then we have to check whether the same patch can be obtained by a construction corresponding to a lexicographically smaller string (in this case we reject the map, otherwise we accept it). To this end we check for every edge of the map in turn whether it is the starting edge of a cut path and if yes (that is: it encodes a construction) we compute its string. Since we do not distinguish between mirror images, we also have to check whether the mirror images can be constructed in a way corresponding to a smaller string—that is: we have to run the same checks for all edges with the role of left and right interchanged and taking σ as clockwise orientation when computing the string.

Since for a given edge and orientation the string can obviously be computed in linear time, we get:

Lemma 2. *The canonicity test just described has time consumption quadratic in the number of vertices.*

Obviously the correctness of the method is not affected by adding redundant information to the string, that is: information that is encoded in the string anyway. So one can e.g. give the size of the face on the right-hand side (left-hand side when working in the mirror image) of the starting edge as the first entry and only after that the string described above. This way edges with a larger face on the right-hand side need not be checked in the non-mirror image run (the string corresponding to the construction would be larger anyway). This reduces the number of edges that have to be looked at and in case an edge with a smaller face on the right-hand side than coded in the string corresponding to the construction is found to be a starting edge of a cut path (in general this can be done in sublinear time), there is no need to compute the whole string—it will be lexicographically smaller anyway.

Analogously we can number the possible cut paths, say 1 for a Jordan curve path, 2 for a dumb-bell path, 3 for a sandwich path and 4 for a figure 8 path and add the type of the path as a first entry to the string. Again in a lot of cases it can be avoided to compute the complete string, since when we have determined whether an edge is a starting edge of a cut path at all, we also know the type of the path. Some of these heuristics are used in the programs—always depending on the case. Though they do not change the worst case analysis of the running time, they have a large effect on the practical performance of the programs. For details see [12,13].

7. Part 6: reducing the number of patches to store

The main problem to be solved is to avoid the generation and listing of patches that will never be glued into a path to form a map (we call this *direct use*) or be composed with other patches to form a patch that is directly used (we call this *indirect use*). We could not find a fast computable necessary *and* sufficient condition for patches to be of direct or indirect use—and judging from the literature we doubt that such a criterion

does exist. We used some easily and fast computable only necessary conditions that sort out quite a number of useless patches. For a detailed treatment of the effect of these criterions on the number of patches, see [12,13]. We will only describe the two most successful conditions used.

Due to the Euler formula, the number g of faces in a map with n vertices is $g = (n/2)+2$ in the cubic and $g = n+2$ in the quartic case. The basic idea is to compute a lower bound for the number of faces in a map containing the patch and reject the patch in case this number is larger than g. For a patch P let $f(P)$ denote the number of faces contained in P and $k(P)$ the number of vertices of valency 2 in its boundary.

The most powerful criterion in the quartic case is the following

Lemma 3. *All maps except the Octahedron can be constructed by just using patches P with the property that $f(P)+k(P)+5 \leqslant g$.*

Proof. This can be shown by observing that though $k(P)$ might decrease when gluing together patches, the value of $f(P)+k(P)$ is monotonically increasing. The three patches used in a sandwich, figure 8 or dumb bell path have $k(P) \in \{1,2\}$, so $f(P)+k(P)+5 > g$ would require the remaining 2 patches to have at most 5 ($k(P)=1$), respectively, 6 ($k(P)=2$) faces altogether, which can easily be shown to be impossible by looking at small patches.

In case of a Jordan curve path, the remaining patch would have to have only 4 faces. This is indeed possible, but the Octahedron is the only quartic planar map with only Jordan curve paths and all the Jordan curve paths split off a patch with 4 faces. So all other maps can also be built without such patches. Of course in the isomorphism rejection routine it must be made sure that no paths based on a Jordan curve path that splits off a patch with 4 faces is used to reject the structure. □

The most powerful tool in the cubic case is:

Lemma 4. *Let s be the degree of the smallest face in a map G (so $s \in \{3,4,5\}$), P a patch contained in G and $d = 12 + \sum_{f \in P}(deg(f)-6)$.*

Then P fulfills the condition $f(P) + \lceil d/6 - s \rceil \leqslant g$.

Proof. The Euler formula implies that in a cubic planar map with F the set of faces we have $\sum_{f \in F}(6-deg(f)) = 12$. So $\sum_{f \in F \setminus P}(6-deg(f)) = 12 + \sum_{f \in P}(deg(f)-6) = d$. This implies $(g-f(P)) \cdot (6-s) \geqslant d$ and finally $g \geqslant f(P) + d/6 - s$. □

A method that reduces the storage consumption is not to store patches that are so large that they can only be combined with smaller ones (that is: $f(P) > g/2$). All the other patches needed to form larger patches or even maps are already in the list, so we can just glue them together or form maps and delete the patch afterwards. Though this reduces neither the number of patches that are generated nor the running time, it does reduce the number of patches that are kept in the

memory and for some cases the size of the machine memory turned out to be the bottleneck.

8. Part 7: checking the programs

The algorithms described above were implemented as computer programs in C and are called *CPF* (cubic maps) and *ENU* (quartic maps). The source codes can be obtained from the authors.

The results of *ENU* were checked with the help of the program *GENREG* by Markus Meringer (see [17]). *GENREG* generates regular graphs with a given number of vertices. The graphs generated by this program were filtered for planar ones by a program using the Hopcroft–Tarjan algorithm implemented in the program package *LEDA*.[1] For planar graphs this algorithm also provides planar embeddings. For many different combinations of allowed face sizes and vertex numbers up to $v = 18$ the numbers of maps obtained by this procedure were compared with those generated by *ENU*. In cases where maps occurred that were not 3-connected, they were sometimes missing in the embedded output of *GENREG*, because the graph was embedded in a combinatorially different way. These cases were checked by hand.

The program *CPF* was checked against the program *fullgen* [5] for Fullerenes and against the program *plantri* [6]. In order to check *CPF* against *fullgen*, all fullerenes with up to 120 vertices were generated. *Plantri* was used to generate all 3-connected cubic planar maps with up to 32 vertices. For every map the set of face degrees has been determined. The resulting statistics has been used to check *CPF* on about 200 parameter sets.

In all cases we checked, we had complete agreement.

9. Part 8: results

This section contains some results obtained by these programs. Let $C_{n,D}$ ($Q_{n,D}$) denote the set of all cubic (quartic) planar maps which have n vertices and the set of face degrees is exactly D (and not just a subset of D). Gaps between two consecutive vertex numbers in the tables and dashes denote that the Euler formula or a result by Grünbaum [11] implies that no maps exist.

The number of structures generated per second as well as the memory consumption depends strongly on the set of parameters. If the allowed face sizes are small (e.g. at most 7-gons in the cubic case or at most 5-gons in the quartic case) the programs are much faster and need much less memory than in cases where large faces are allowed.

Sample running times, memory consumption for the patches and rounded generation rate on a 350 MHZ Pentium II with Linux operating system are: 12 min/96 kbytes

[1] Library of efficient data structures and algorithms, http://www.mpi-sb.mpg.de/LEDA/leda.html.

for $Q_{95,\{3,4\}}$ (64 structures/s), 9.5 min/26 Mbytes for $Q_{23,\{3,4,5\}}$ (327 structures per second), 43 s/14 Mbytes for $Q_{30,\{3,9\}}$ (1 structure/s), 5 min/136 Mbytes for $Q_{30,\{3,7\}}$ (15 structures/min), 6.5 s/131 kbytes for $C_{200,\{3,6\}}$ (2.6 structures/s), 13 min/5 Mbytes for $C_{80,\{3,4,5,6\}}$ (98 structures/s), 5.7 h/1.3 Mbytes for $C_{68,\{5,9\}}$ (8.8 structures/h), 47.7 s/200 kbytes for $C_{70,\{5,6\}}$ (170 structures/s).

n	$\lvert Q_{n,\{3,4\}}\rvert$	n	$\lvert Q_{n,\{3,4\}}\rvert$	n	$\lvert Q_{n,\{3,4\}}\rvert$	n	$\lvert Q_{n,\{3,4\}}\rvert$
10	2	23	33	36	499	49	1554
11	1	24	76	37	366	50	2505
12	5	25	51	38	650	55	2829
13	2	26	109	39	493	60	6234
14	8	27	78	40	815	65	6631
15	5	28	144	41	623	70	13428
16	12	29	106	42	1083	75	14021
17	8	30	218	43	800	80	26257
18	25	31	150	44	1305	85	26228
19	13	32	274	45	1020	90	47928
20	30	33	212	46	1653	95	46518
21	23	34	382	47	1261	100	81084
22	51	35	279	48	2045	105	77795

n	$\lvert Q_{n,\{3,4,5\}}\rvert$	$\lvert Q_{n,\{3,5\}}\rvert$	$\lvert Q_{n,\{3,4,6\}}\rvert$	$\lvert Q_{n,\{3,6\}}\rvert$	$\lvert Q_{n,\{3,5,6\}}\rvert$
10	0	1	0	—	0
11	2	—	0	—	0
12	5	0	0	3	0
13	11	—	2	—	0
14	36	2	9	—	0
15	74	—	20	0	3
16	232	0	60	—	8
17	539	—	106	—	2
18	1576	10	304	16	0
19	4014	—	669	—	19
20	11489	2	1836	—	114
21	30622	—	4446	0	153
22	87043	52	11804	—	300
23	238007	—	30050	—	492
24	673547	31	80896	224	1616

n	$\lvert C_{n,\{3,6\}}\rvert$	n	$\lvert C_{n,\{3,6\}}\rvert$	n	$\lvert C_{n,\{3,6\}}\rvert$	n	$\lvert C_{n,\{3,6\}}\rvert$
8	1	60	6	112	13	164	8
12	2	64	9	116	6	168	18
16	3	68	4	120	14	172	9
20	2	72	8	124	7	176	17
24	3	76	5	128	15	180	16
28	3	80	10	132	10	184	13
32	5	84	8	136	10	188	9
36	4	88	7	140	10	192	28
40	4	92	5	144	20	196	12
44	3	96	15	148	8	200	17
48	8	100	7	152	11	204	14
52	4	104	8	156	12	208	20
56	5	108	9	160	20	212	10

n	$\lvert C_{n,\{4,6\}}\rvert$	n	$\lvert C_{n,\{4,6\}}\rvert$	n	$\lvert C_{n,\{4,6\}}\rvert$	n	$\lvert C_{n,\{4,6\}}\rvert$
12	1	60	19	108	79	156	202
14	1	62	21	110	89	158	237
16	1	64	22	112	97	160	262
18	1	66	16	114	68	162	175
20	3	68	36	116	133	164	330
22	1	70	21	118	90	166	239
24	3	72	29	120	99	168	249
26	3	74	31	122	115	170	288
28	3	76	34	124	127	172	319
30	2	78	24	126	86	174	209
32	8	80	53	128	171	176	397
34	3	82	32	130	118	178	291
36	7	84	42	132	133	180	298
38	7	86	47	134	152	182	356
40	7	88	50	136	165	184	388
42	5	90	35	138	110	186	259
44	14	92	75	140	220	188	479
46	6	94	46	142	150	190	352
48	12	96	59	144	164	192	352
50	12	98	65	146	189	194	418
52	13	100	70	148	207	196	463
54	10	102	48	150	142	198	303
56	23	104	99	152	265	200	559
58	12	106	65	154	190	202	419

n	$\lvert C_{n,\{3,4,7\}}\rvert$	$\lvert C_{n,\{3,5,7\}}\rvert$	$\lvert C_{n,\{3,4,5,7\}}\rvert$	$\lvert C_{n,\{3,4,8\}}\rvert$	$\lvert C_{n,\{3,5,8\}}\rvert$	$\lvert C_{n,\{3,4,5,8\}}\rvert$
8	0	0	—	0	—	—
10	1	—	0	0	1	0
12	3	2	1	2	0	0
14	0	—	8	2	0	4
16	7	1	15	0	6	17
18	7	—	34	1	1	36
20	0	20	83	13	15	66
22	24	—	203	16	21	238
24	32	15	551	3	5	878
26	14	—	1328	38	136	2127
28	158	284	2979	180	151	5913
30	101	—	8547	180	180	19323
32	179	389	22026	156	1932	64072
34	841	—	55975	1408	839	193044
36	570	5309	141162	3363	6326	579388
38	1500	—	379349	3897	19589	1873193
40	6163	11428	986917	8882	11548	?

n	12	20	28	36	44	52						
$\lvert C_{n,\{3,7\}}\rvert$	0	4	0	29	1	638						
n	14	24	34	44								
$\lvert C_{n,\{3,8\}}\rvert$	1	6	19	298								
n	8	12	16	20	24	28	32	36	40	44	48	
$\lvert C_{n,\{3,9\}}\rvert$	0	1	1	3	2	9	14	59	145	559	1845	
n	14	20	26	32	38	44	50	56				
$\lvert C_{n,\{4,7\}}\rvert$	1	2	4	12	26	127	431	2189				
n	16	24	32	40	48	56						
$\lvert C_{n,\{4,8\}}\rvert$	2	5	32	174	1710	18897						
n	18	28	38	48								
$\lvert C_{n,\{4,9\}}\rvert$	1	3	11	109								
n	28	32	36	40	44	48	52	56	60	64	68	72
$\lvert C_{n,\{5,7\}}\rvert$	1	0	2	0	13	0	73	5	620	81	6556	1675
n	32	38	44	50	56	62	68					
$\lvert C_{n,\{5,8\}}\rvert$	1	1	3	5	27	58	314					
n	36	44	52	60	68							
$\lvert C_{n,\{5,9\}}\rvert$	1	0	4	0	50							

References

[1] D. Babic, N. Trinajstic, Resonance energies of fullerenes with 4-membered rings, Internat. J. Quantum Chem. 55 (1995) 309–314.
[2] G. Brinkmann, Problems and scope of the spiral algorithm and spiral codes for polyhedral cages, Chem. Phys. Lett. 272 (3–4) (1997) 193–198.
[3] G. Brinkmann, Isomorphism rejection in structure generation programs, in: P. Hansen, P.W. Fowler, M. Zheng (Eds.), Discrete Mathematical Chemistry, Vol. 51, DIMACS Series on Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, 2000, pp. 25–38.
[4] G. Brinkmann, M. Deza, Lists of face-regular polyhedra, J. Chem. Inform. Comput. Sci. 40 (2000) 530–541.
[5] G. Brinkmann, A.W.M. Dress, A constructive enumeration of fullerenes, J. Algorithms 23 (1997) 345–358.
[6] G. Brinkmann, B.D. McKay, Fast generation of non-isomorphic planar cubic graphs, see http://cs.anu.edu.au/∼bdm/index.html, in preparation.
[7] H.S.M. Coxeter, Regular Polytopes, Dover, New York, 1973.
[8] P.W. Fowler, T. Heine, D.E. Manolopoulos, D. Mitchell, G. Orlandi, G. Seifert, R. Schmidt, F. Zerbetto, Energetics of fullerenes with four-membered rings, J. Phys. Chem. 100 (1996) 6984–6991.
[9] P.W. Fowler, D.E. Manolopoulos, An Atlas of Fullerenes, Oxford University Press, Oxford, 1995.
[10] O.D. Friedrichs, A.W.M. Dress, A. Müller, M.T. Pope, Polyoxometalates: a class of compounds with remarkable topology, Mol. Eng. 3 (1993) 9–28.
[11] B. Grünbaum, Convex Polytopes, Wiley, London, New York, Sydney, 1967.
[12] T. Harmuth, Die Generierung simpler, 3-regulärer planarer, zusammenhängender Graphen mit vorgegebenen Flächengrößen, Diplomarbeit, Universität Bielefeld, 1997.
[13] O. Heidemeier, Die Erzeugung von 4-regulären, planaren, simplen, zusammenhängenden Graphen mit vorgegebenen Flächentypen, Diplomarbeit, Universität Bielefeld, 1998.
[14] H.W. Kroto, J.R. Heath, S.C. O'Brien, R.F. Curl, R.E. Smalley, C_{60}: buckminsterfullerene, Nature 318 (1985) 162–163.
[15] J. Malkevitch, Polytopal graphs, in: L.W. Beineke, R.J. Wilson, (Eds.), Selected Topics in Graph Theory, Vol. 3, 1998, pp. 169–188.
[16] D.E. Manolopoulos, J.C. May, S.E. Down, Theoretical studies of the fullerenes: C_{34}–C_{70}, Chem. Phys. Lett. 181 (2,3) (1991) 105–111.
[17] M. Meringer, Fast generation of regular graphs and construction of cages, J. Graph Theory 30 (2) (1999) 137–146.