A Constructive Enumeration of Fullerenes

Gunnar Brinkmann and Andreas W.M. Dress

Fakultät für Mathematik

Universität Bielefeld

D 33501 Bielefeld Germany

gunnar@mathematik.uni-bielefeld.de *dress@mathematik.uni-bielefeld.de*

Proposed running head: A Constructive Enumeration of Fullerenes

Proofs shall be sent to:

Gunnar Brinkmann
Fakultät für Mathematik
Universität Bielefeld
D 33501 Bielefeld

Abstract

In this paper, a fast and complete method to enumerate fullerene structures is given. It is based on a top-down approach, and it is fast enough to generate, for example, all 1812 isomers of C_{60} in less than 20 seconds on an SGI-workstation. The method described can easily be generalised for 3-regular spherical maps with no face having more than 6 edges in its boundary.

Symbols:

$\mathcal{P}$: A calligraphic P

Introduction

A fullerene is a spherically shaped molecule built from carbon atoms so that every carbon ring is either a pentagon or a hexagon, and every atom has bonds with exactly 3 other atoms. Consequently, a combinatorial fullerene is a 3-regular spherical map having pentagonal and hexagonal faces, only. For such structures, the Euler formula implies that there are always 12 pentagons and $h = n/2 - 10$ hexagons (with n the number of vertices).

Ever since the first fullerene – the famous C_{60} or "buckyball" – was discovered in 1985 (c.f. [7]), many attempts have been made to generate large lists of combinatorial fullerenes (see e.g. [9],[11], or [8]). Nevertheless, the proposed methods either were not complete (they could not guarantee that *every possible* structure is generated), or were not efficient enough to be able to generate lists of interesting size, – or both.

Already existing mathematical approaches towards enumerating three-regular spherical maps (see [1] or [12]) could not be used to obtain fast algorithms for this restricted class of structures.

In this paper, a new approach will be discussed in enough detail so that – in principle – every reader could implement it as a computer program. A first rough sketch of the new method, omitting isomorphism testing, coding and other important details that make the implementation efficient, has been given in [3].

We assume the reader to be familiar with the basic definitions and notations in Graph Theory.

Part 1: The Algorithm

The algorithm will be described in a top down manner. That is: We start with the coarsest step and proceed to the finer ones, ending up with the details that make its implementation as a computer program efficient.

Breaking a Fullerene into Parts

If we walk along an edge in a planar embedding of a 3-regular graph and approach a vertex v, then it is well defined which of the the two edges adjacent to v (different from the one we are walking along) is on the right and which one is on the left. Starting at an arbitrary vertex and walking along one of the three edges adjacent to it, we can choose an alternating path by taking the right and the left direction alternatively whenever we approach a new vertex. Such a path is called a *Petrie* path (c.f. [5]).

Suppose the edges we are walking along are the edges $e_1, e_2, \ldots$. Since our graph is finite, there is some smallest $n > 0$, so that (independent of direction) $e_{n+1} = e_{n'}$ for some index n' with $1 \leq n' < n$. If $e_{n+1} = e_1$ and $e_{n+2} = e_2$, we have a closed Petrie path (a Jordan curve path) cutting the spherical graph into two hemispheres, both of which have a zig-zag boundary with precisely n edges. This is illustrated in Figure 1.

If this is not the case, we can reverse our direction: We start at e_2, turn left or right to walk along e_1 when we approach the common vertex of e_1 and e_2 and after that again turn right and left alternatively walking along the edges $e_0, e_{-1}, \ldots$ until again we have some $e_{-(m+1)}$ with

$e_{-(m+1)} = e_{m'}$ with $-m < m' \leq n$.

In this case, the union $\{e_{-m}, e_{-m+1}, \ldots, e_n\}$ of the edges of the Petrie path starting at e_{-m} and ending with e_n together with their adjacent vertices form a graph with exactly two vertices of valence 3 (the points where we had to stop) and all the remaining vertices of valence 2. Classifying these paths by their homeomorphism classes – corresponding to the two isomorphism classes of graphs consisting of three edges and two vertices of degree 3 – we get two types of paths (cf. Figure 2). They are called *dumb-bell* and *sandwich* path. Both split the planar graph into three parts. So, we do have the following result:

Theorem 1 (and Definition)

Every Fullerene can be cut into two or three parts – called patches *– by appropriately chosen Petrie paths. Up to homeomorphism, there are three different* types *of Petrie paths.*

If, for instance, we rotate one of the patches in Figure 1 by two edges and fix the other, we get a non-isomorphic fullerene. To deal with this fact, we consider *marked* patches, that is we consider patches together with a mark on one of their boundary vertices. Two marked patches are isomorphic if and only if there is an isomorphism from one onto the other that also maps the marks onto each other.

Assuming that we have lists of all possible marked patches that can occur, we can proceed as follows:

First, we enumerate all non-isomorphic Petrie paths that could occur, given the size of a fullerene graph. To obtain an upper bound for the length of a Petrie path we do some simple analysis of the patches we construct (see Part 2). A Jordan curve path is completely described by its number of edges. For the other two types, we must also enumerate all – non-isomorphic – possibilities to distribute the edges onto the 3 segments of the path. This way, the path and also the shapes of the boundaries of the regions separated by the path are completely determined. Now, we can fix an *arbitrary* vertex in the boundary of each region to carry the mark for this region.

For each of these possibilities, we enumerate all combinations of how the (given) number of hexagons can be partitioned into a sum of (two or three) non-negative integers and assign these numbers to the (two or three) regions in every possible way.

Now, we look into our list to find out whether, for each of the (two or three) regions, we can find marked patches with the given shape of the boundary and the given number of hexagons. If yes, they are glued into the path along their boundaries (gluing the marks onto each other) to obtain every possible combination which gives rise to a fullerene.

If our list of marked patches was complete, then we will get a complete list of fullerenes (which – so far – are of course not necessarily pairwise non-isomorphic).

Part 2: Generating the List of Patches

From the viewpoint of patches, an alternating path along the boundary is an alternating sequence of vertices of valence 2 (called 2-vertices) and vertices of valence 3 (3-vertices).

Looking at the structure of the boundary of patches that are cut out by Petrie paths, we observe that the boundary consists of segments of alternating paths, possibly separated by at most four localities where we have edges adjacent to two 2-vertices. In the generation procedure, we will need patches with up to 6 such localities. Patches where never two 3-vertices follow each other are called *pseudo-convex* patches. As will be seen later on, in a pseudo-convex patch the number of pentagons and the number of edges both of whose vertices are 2-vertices always add up to 6.

Observation 1

Petrie paths divide fullerenes into pseudo-convex patches.

The boundary structure of pseudo-convex patches can easily be encoded: Suppose that for $0 \leq k \leq 6$ we have a patch with exactly k edges in the boundary for which both endpoints are 2-vertices, called *break-edges.* If we have a Jordan curve path ($k = 0$), we simply take the number of 3-vertices (or, equivalently, of 2-vertices) to encode its structure. If $k > 0$, we encode the boundary by a sequence of k numbers $a_1, \ldots, a_k$ denoting the number of 3-vertices in between two break-edges in a cyclic (clockwise) order, so that the sequence is lexicographically maximal – look at Figure 3 for some examples.

Euler's formula implies easily that the number of Pentagons in such a patch must be equal to $6 - k$. The task is now to construct, for a given number S of hexagons, a complete list of marked pseudo-convex patches with up to S hexagons. In [3], this task was dubbed a *convex PentHex Puzzle.*

Since we can put the marks on any vertex occuring in our Petrie path, we can always choose a vertex that corresponds to a 2-vertex incident with a break-edge of the given patch (except in the Jordan curve case). Furthermore, among all such vertices, we can choose one lying in clockwise direction of the break-edge, so that evaluating the encoding of the boundary starting at this vertex gives an encoding that is lexicographically maximal among all possible ones. Patches with such a mark are called *canonically marked.* We can restrict the generation procedure to generate canonically marked patches, only. For canonically marked, pseudo-convex patches, we do not allow reflections as isomorphisms, since gluing into the emerging fullerene structure the mirror image of a patch will in general lead to a different fullerene.

Theorem 2

The class $\mathcal{P}$ of all non-isomorphic, canonically marked, pseudo-convex patches can be recursively enumerated without isomorphism tests.

Proof: Starting with a hexagon or pentagon, the patches are constructed by enlarging the initial patch step by step in a canonical way. The method is understood most easily, if we describe it the other way round: We describe how a given patch in $\mathcal{P}$ is uniquely *reduced* to a smaller patch from $\mathcal{P}$. The reverse of the described operations can then be used to *construct* the required set. Every marked patch has a unique (smaller) *ancestor*, but in general every marked patch can be enlarged in more than one way to obtain larger marked patches of which it is the ancestor, that is, it can have more than one *offspring* patch.

So, suppose we are given a canonically marked patch P. We have to describe how a unique ancestor in $\mathcal{P}$ can be obtained:

First look at Figure 4 where a patch without any break-edges is given ($k = 0$). The set of faces sharing an edge with the boundary form a *layer* – i.e. every face in this set has a connected intersection with the boundary (in this case two adjacent edges) and the subgraph induced by these faces in the inner dual of our patch is a simple cycle. An easy way to prove this fact is to observe that in case one face in this set has a disconnected intersection with the boundary or two faces are adjacent in the inner dual and their union has a disconnected intersection with the boundary (in which case the subgraph of the inner dual induced by the faces in the layer would not be a simple cycle), removing these faces would disconnect the boundary of the patch as well as the patch itself (by the Jordan Curve theorem). Trying to choose these faces in a way that one of the parts has minimum size and using the fact that we only have pentagons and hexagons immediately will lead to a contradiction (see Figure 5).

So we can just remove this layer. The new mark is put on that canonical new 2-vertex that is closest to the old mark in counterclockwise direction. The new patch contains fewer faces (a_0 faces are removed) and can easily be shown to have no two adjacent 3-vertices in its boundary. If only hexagons were removed, the boundary encoding stays the same – otherwise we obtain an encoding of type $k > 0$, with k the number of pentagons removed. In any case, we obtain a smaller patch that is also in $\mathcal{P}$.

As an example, it is shown in Figure 6 how a patch with $k > 0$ is reduced, step by step, to a hexagon: We first remove the face that contains the marked vertex. Then we proceed to remove boundary faces in clockwise direction until we obtain a (smaller) pseudo-convex patch. This will always be the case after a finite number of steps, namely when – for the first time – we remove a pentagon or another face containing a break-edge, or when we have removed a complete layer (in case $k = 1$). Again, it needs some proof – yet, it is easy to see – that the resulting patch must be connected, as we again suppose the patch to contain hexagons and pentagons, only (in fact, the proof can follow the lines sketched above). The new mark is again put on that canonical new 2-vertex that is closest to the old mark in counterclockwise direction.

Consequently, for constructing the set $\mathcal{P}$, we start (a) with a hexagon and (b) with a pentagon and, in every step, we apply all operations that are inverse to the described reduction steps, until the number of hexagons in the patches exceeds S. ■

Part 3: Storing the Patches

In order to speed up the resulting algorithm, it is essential to determine in a very efficient way whether, for a given boundary encoding and a given number of hexagons, we can find patches which we can glue into our Petrie path. In case there are such patches, they must be easily accessible. The way the patches are encoded and stored is a detail of the *implementation*. Yet, as this detail is *essential* for the efficiency of the algorithm and, in addition, shows us an even more efficient way to generate the patches with 6 pentagons, we will describe it shortly.

First it can be seen easily that not all pseudo-convex patches can occur as patches of Fullerenes: indeed it follows immediately from our Petrie-path construction that no patches with $k > 4$ or with more than two non-zero entries in the boundary encoding can occur. So such patches do not have to be stored once all their offspring patches have been generated.

The remaining patches are stored in a tree like data structure. The branching at the root is done according to the number of hexagons in the patch. After that, the branching at level i is done according to the i-th entry of the boundary sequence. At every node of the tree, we have a (possibly empty) linear list of patches with the number of hexagons and the boundary sequence given by the (unique) way from the root to the node. So, if we are looking for a patch with 20 hexagons and with boundary encoding $6, 2, 4$, we take the 20-th branch at the root, the 6-th branch at level one, the 2-nd branch at level two, and the 4-th branch at level three. Then, we check whether a nonempty list of patches is stored at this node. This way, we can determine very efficiently, whether or not a region in a pregiven path can be filled with a certain number of hexagons.

For enumerating all fullerenes with 170 vertices, we generated 11 342 885 marked patches – 9 968 867 of them with 6 pentagons (needed only for the Jordan curve case). In order to access such a large list very fast we have to keep it in the main memory of the machine. Hence, the precise structure of the patches themselves has to be encoded in a very efficient manner, too.

Theorem 3

Pseudo-convex patches consisting of (altogether n) pentagons and hexagons, can be stored in an array (of integers none exceeding n) of constant size.

Proof: We use a spiral code for the encoding: The same argument that shows that the reduction method used in Part 2 works, can be used to show that pseudo-convex patches with at least one break-edge ($k > 0$) can be unwound in a spiral *from the outside* starting at the mark, that is: we first remove the face with the mark and then proceed to remove faces in clockwise direction along the boundary (note the difference to the spiral ring algorithm described e.g. in [9]). This way, all the faces will be removed, one by one. We obtain an efficient code, if we just store the places where pentagons occur (there are at most $p = 5$ pentagons, so we need at most 5 entries per patch). Given the shape of the boundary (encoded in the boundary sequence that needs at most $6 - p$ entries), the patch can obviously be reconstructed easily. Look at Figure 7 for an example.

In the case of 6 pentagons ($k = 0$) we have a completely symmetric boundary. For every unmarked patch, the number N of marked patches is the length of the boundary divided by the order of its (orientation preserving) automorphism group. All N non-isomorphic marked patches can be found by putting a mark on every of N subsequent 2-vertices. Furthermore, any patch without pentagons in its boundary can be obtained from one with pentagons in its boundary (called its *kernel*) by adding layers of hexagons. It has the same automorphism group as its kernel. So we just have to store the (arbitrarily marked) kernel, the order of its automorphism group, and the number of hexagon layers.

Starting at a boundary pentagon, kernels can also be unwound in a spiral fashion from the outside, so they can also be stored using an inverse spiral code. This means that in addition to the boundary encoding, the number of layers and the order of the automorphism group (3 entries), we need another 5 entries for the position of the pentagons (the first pentagon is always on position 1). ■

In our implementation, we put the mark on a boundary pentagon in the kernel in such a way that the spiral development starting at that mark is lexicographically minimal. This minimal spiral development of the kernel is called its *canonical form* and all the kernels are stored in their canonical form. This will be needed for the isomorphism test of fullerenes. While calculating the canonical form, we also obtain the order of the automorphism group.

These observations enable us to construct patches of type $k = 0$ in an even more efficient way than the one described in Section 2:

First, it is easily seen that every patch of type $k = 0$ with at least one pentagon in its boundary – that is, every kernel – can be obtained from a patch of type $k = 2$ by adding one row of hexagons with 2 pentagons at its ends or from a patch ot type $k = 1$ by adding a layer containing exactly 1 pentagon. The patches of type $k = 0$ without pentagons in the boundary can then simply be obtained by adding an entry for the number of layers of hexagons to the code and leaving the coding of the kernel and the size of the automorphism group unchanged.

Part 4: Testing for isomorphisms

For the list of all non-isomorphic fullerenes on 170 vertices, from the 11 342 885 patches mentioned above, our algorithm produced altogether 1 802 856 946 fullerenes, 46 088 148 of which were non-isomorphic. Because of the large number of non-isomorphic graphs, it is impossible to store them – and even if one could store them, it would take an enormous amount of time to search the (growing) list for isomorphic copies each time a new structure is generated. So, we have to be able to decide for each structure we generate at once – without comparing it to already generated ones – whether we need it in our list or not. To this end, we define a *canonicity criterion*. It has the property that for each isomorphism class of structures, **exactly** one *canonical representative* is generated.

The large number of generated structures and the fact that we deal with (compared to most other generation problems, cf. [4], [6], [10]) very large structures – more than 210 vertices for the largest ones –, make it essential that the canonicity test is very efficient. Using the test described in the sequel, on a HP 9000/735 about 7000 graphs on 170 vertices can be tested per second.

Theorem 4

There is a canonicity check for the fullerenes generated by the algorithm with time consumption (per structure) linear in the number of vertices.

Proof: First, observe that every fullerene has a Petrie path with the first two edges e_1 and e_2 being boundary edges of a pentagon. Whenever a new fullerene is generated, we first check whether the path it is constructed from is one that can be generated starting from such edges e_1 and e_2. If not, the structure is rejected. This can obviously be done in time linear in the length of the path.

Otherwise, we associate a code to the fullerene that describes the *generation history* (compare [2]) of the fullerene. The first entry of the code is the number of edges contained in the Petrie path. The second number describes the kind of path: 1 for a Jordan curve path, 2 for a dumb-bell path, and 3 for a sandwich type path. The rest of the code is defined depending on the type of path. The code will only be described for the dumb-bell path, but is very similar for the other two cases:

The dumb-bell path has three parts, as shown in Figure 8. The path is constructed as a directed path with a well defined beginning and end. We assume that Part 1 is at least as large as

Part 3 and that the turn from the first edge to the second edge is a left turn (note that we do not want to distinguish between mirror images of Fullerenes). The number of edges in Part 1 and the number of edges in Part 3 are the next entries for our code. Finally, the spiral codes of the marked patches are appended to the code of the fullerenes – first the code of the patch surrounded by Part 1 of the path, then that surrounded by Part 3, and finally that patch that is adjacent to all 3 parts.

Having determined the code of the fullerene, we try to construct all paths that can be obtained with e_1 and e_2 being adjacent edges of a pentagon and try to reconstruct the fullerene from each of these paths. There are at most 120 of them – although in almost all cases there are much less, since, in general, different edge pairs will lead to the same path. Each reconstruction corresponds to a new code. We accept the fullerene if the original code is the lexicographically smallest one among all reconstructed codes, and we discard it otherwise. Due to our assumption that we first turn left, we have to calculate the code in the mirror image of the fullerene if in the constructed path the first turn is right.

Since there are at most 120 paths to be constructed and since the associated code can be derived in linear time, the time consumption of the canonicity test is linear in the number of vertices.

■

Nevertheless, for the actual performance of the program, even more important than the worst case analysis is the fact that a fullerene can be rejected already, whenever a shorter path is found. If no shorter path and no path of the same length with a smaller type index is found, the full code has to be reconstructed only for those paths with the same length and the same type as the original one.

Part 5: Results

Table 1 gives the numbers of all fullerenes generated so far. IPR-fullerenes are fullerenes where no two pentagons share an edge (**I**solated **P**entagon **R**ule). They are supposed to be more stable chemically and, therefore, to occur more likely in nature. Among other questions resulting from this list, the significant similarity between the number of fullerenes with n vertices and that of IPR-fullerenes with $n + 48$ vertices will be a subject of further research.

References

[1] D. Barnette. On generating planar graphs. *Discrete Mathematics*, 7:199–208, 1974.

[2] G. Brinkmann. Generating cubic graphs faster than isomorphism checking. Preprint SFB 343 No. 92-047, 1992. Bielefeld.

[3] G. Brinkmann and A.W.M. Dress. PentHex Puzzles – a reliable and efficient top-down approach to fullerene-structure enumeration. to appear in PNAS.

[4] G. Brinkmann, B.D. McKay, and C. Saager. The smallest cubic graphs of girth 9. *Combinatorics, Probability and Computing*, 5:1–13, 1995.

[5] H.S.M. Coxeter. *Regular Polytopes*. Dover Publications, Inc, 1973.

[6] R. Grund, A. Kerber, and R. Laue. MOLGEN – ein Computeralgebrasystem für die Konstruktion molekularer Graphen. *match 27*, pages 87–131, 1992.

[7] H.W. Kroto, J.R. Heath, S.C. O'Brien, R.F. Curl, and R.E. Smalley. C_{60}: Buckminsterfullerene. *Nature*, 318:162–163, 1985.

[8] X. Liu, D.J. Klein, T.G. Schmalz, and W.A. Seitz. Generation of carbon cage polyhedra. *J. of Comp. Chemistry Vol. 12 No. 10*, pages 1252–1259, 1991.

[9] D.E. Manolopoulos, J.C. May, and S.E. Down. Theoretical studies of the fullerenes: C_{34} to C_{70}. *Chemical Physics letters*, 181, No 2,3:105–111, 1991.

[10] Brendan D. McKay and Gordon F. Royle. Constructing the cubic graphs on up to 20 vertices. *Ars Combinatoria 21 a*, pages 129–140, 1986.

[11] Chih-Han Sah. Combinatorial construction of fullerene structures. *Croatica Chemica Acta*, pages 1–12, 1993.

[12] E. Steinitz and H. Rademacher. Vorlesungen über die Theorie der Polyeder. Berlin, 1934.

Table 1: Numbers of fullerenes and IPR-fullerenes

number of atoms	number of fullerenes	number of atoms	number of IPR-fullerenes
		60	1
		62	0
		64	0
		66	0
20	1	68	0
22	0	70	1
24	1	72	1
26	1	74	1
28	2	76	2
30	3	78	5
32	6	80	7
34	6	82	9
36	15	84	24
38	17	86	19
40	40	88	35
42	45	90	46
44	89	92	86
46	116	94	134
48	199	96	187
50	271	98	259
52	437	100	450
54	580	102	616
56	924	104	823
58	1 205	106	1 233
60	1 812	108	1 799
62	2 385	110	2 355
64	3 465	112	3 342
66	4 478	114	4 468
68	6 332	116	6 063
70	8 149	118	8 148
72	11 190	120	10 774
74	14 246	122	13 977
76	19 151	124	18 769
78	24 109	126	23 589
80	31 924	128	30 683
82	39 718	130	39 393
84	51 592	132	49 878
86	63 761	134	62 372
88	81 738	136	79 362
90	99 918	138	98 541

number of atoms	number of fullerenes	number of atoms	number of IPR-fullerenes
92	126 409	140	121 354
94	153 493	142	151 201
96	191 839	144	186 611
98	231 017	146	225 245
100	285 914	148	277 930
102	341 658	150	335 569
104	419 013	152	404 667
106	497 529	154	489 646
108	604 217	156	586 264
110	713 319	158	697 720
112	860 161	160	836 497
114	1 008 444	162	989 495
116	1 207 119	164	1 170 157
118	1 408 553	166	1 382 953
120	1 674 171	168	1 628 029
122	1 942 929	170	1 902 265
124	2 295 721	172	2 234 133
126	2 650 866	174	2 601 868
128	3 114 236	176	3 024 383
130	3 580 637	178	3 516 365
132	4 182 071	180	4 071 832
134	4 787 715	182	4 690 880
136	5 566 948	184	5 424 777
138	6 344 698	186	6 229 550
140	7 341 204	188	7 144 091
142	8 339 033	190	8 187 581
144	9 604 410	192	9 364 975
146	10 867 629	194	10 659 863
148	12 469 092	196	12 163 298
150	14 059 173	198	13 809 901
152	16 066 024	200	15 655 672
154	18 060 973	202	17 749 388
156	20 558 765	204	20 070 486
158	23 037 593	206	22 606 939
160	26 142 839	208	25 536 557
162	29 202 540	210	28 700 677
164	33 022 572	212	32 230 861
166	36 798 430	214	36 173 081
168	41 478 338		
170	46 088 148		

Captions:

Figure 1: A Jordan Curve Petrie path

Figure 2: A *dumb-bell* Petrie path and a *sandwich* Petrie path

Figure 3: The three patches needed to construct the fullerene in Figure 2 relative to the second marked Petrie path. Break-edges are marked by a thick line. The boundary codes are (9,0),(6,5), and (8,0), respectively.

Figure 4: A patch with 6 pentagons

Figure 5: Two impossible situations for the case $k = 0$

Figure 6: Reduction of a patch to a hexagon in 5 steps

Figure 7: The *inverse spiral code* of a patch

Figure 8: A dumb-bell path with 22 edges in part 1, 7 edges in part 2 and 14 edges in part 3

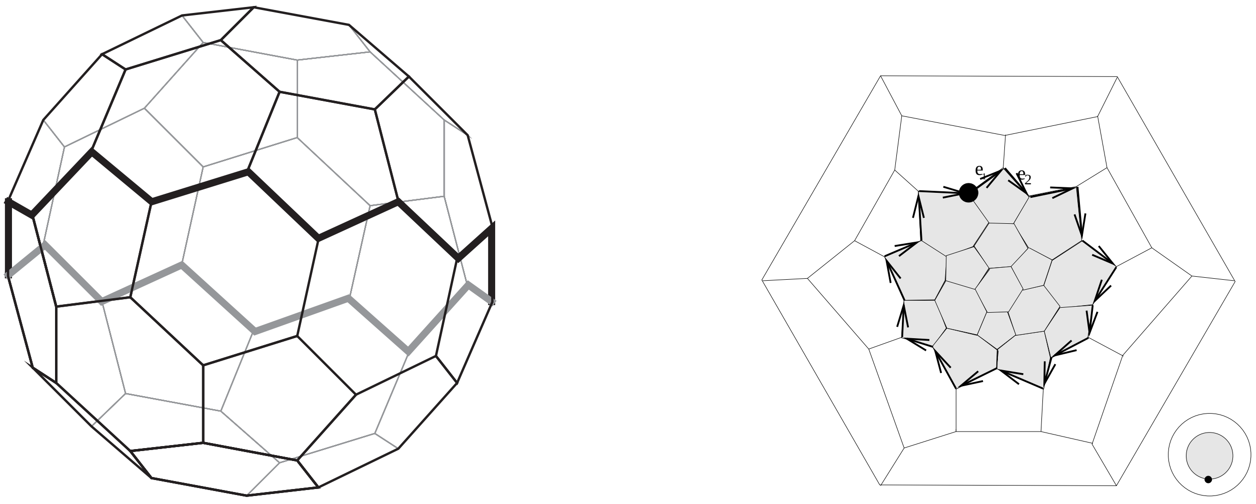

Figure 1

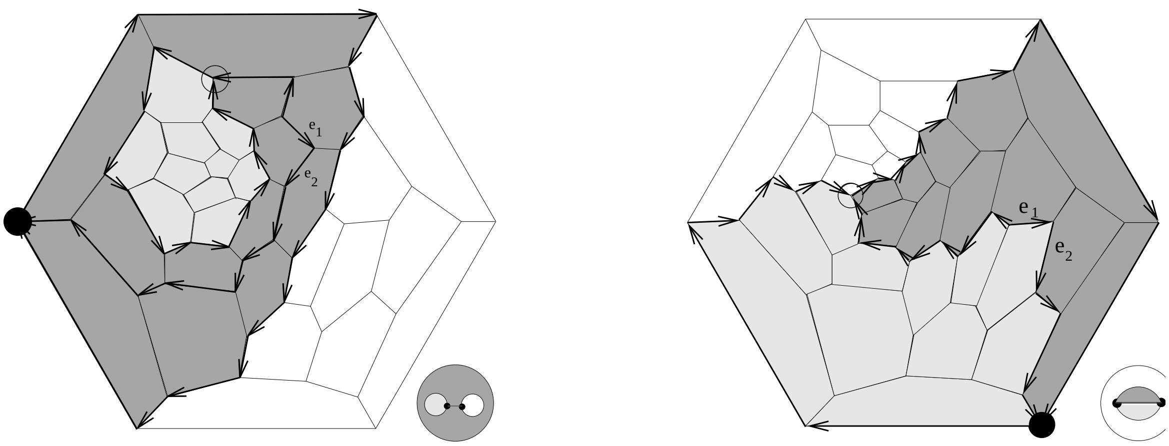

Figure 2

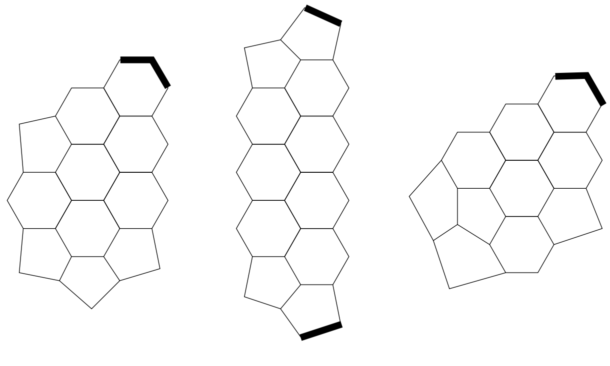

Figure 3

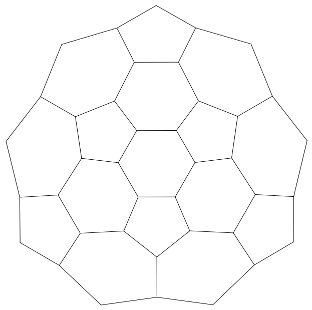

Figure 4

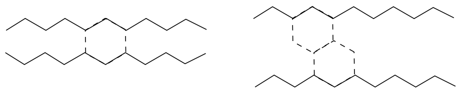

Figure 5

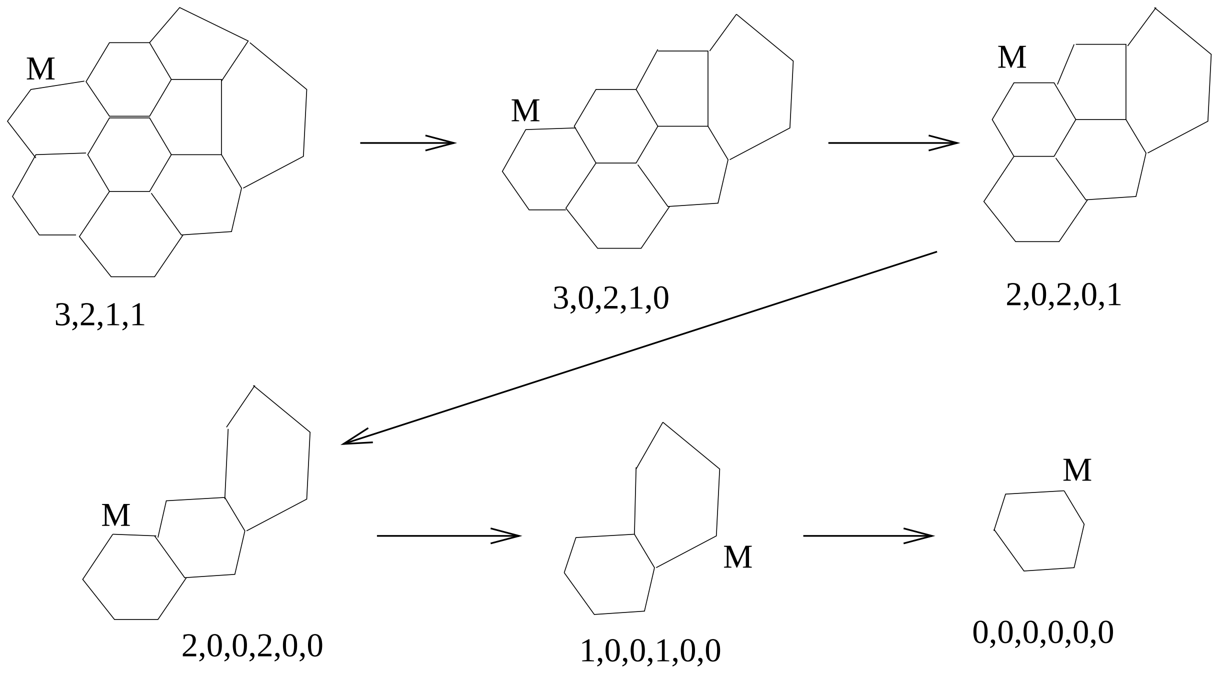

Figure 6

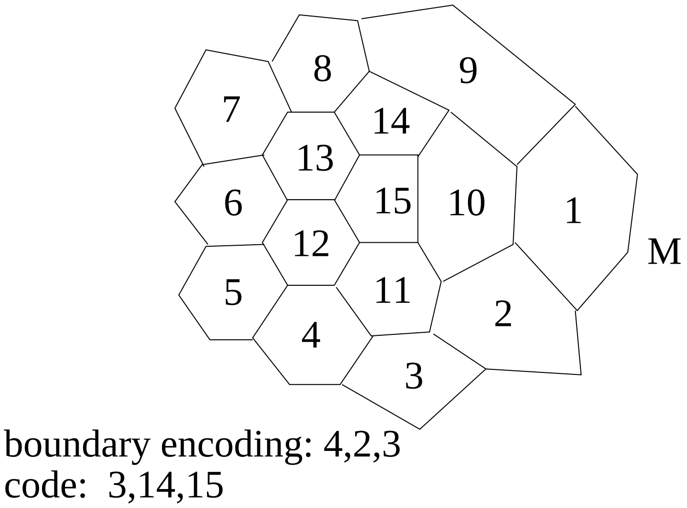

Figure 7

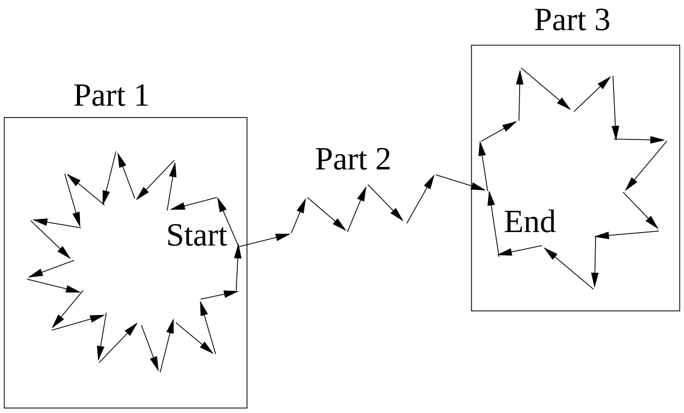

Figure 8