Cover page

Exam information
NFYK10020E - Physics Thesis 60 ECTS, Niels Bohr Institute - Kontrakt:114864 (Frank de Morrée)

Handed in by
Frank de Morrée
fsd776@alumni.ku.dk

Exam administrators
Eksamensteam, tel 35 33 64 57
eksamen@science.ku.dk

Assessors
James Emil Avery
Examiner
j.avery@nbi.ku.dk

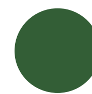
Bjarke Jørgensen
Co-examiner
bjarke@newtec.dk

Hand-in information

Titel: Unfolding Carbon: Algorithmic Generation of Planar Fullerene Precursor Molecules using Intrinsic Geometry
Titel, engelsk: Unfolding Carbon: Algorithmic Generation of Planar Fullerene Precursor Molecules using Intrinsic Geometry
Tro og love-erklæring: Yes
Indeholder besvarelsen fortroligt materiale: No

FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

MSc. in Physics

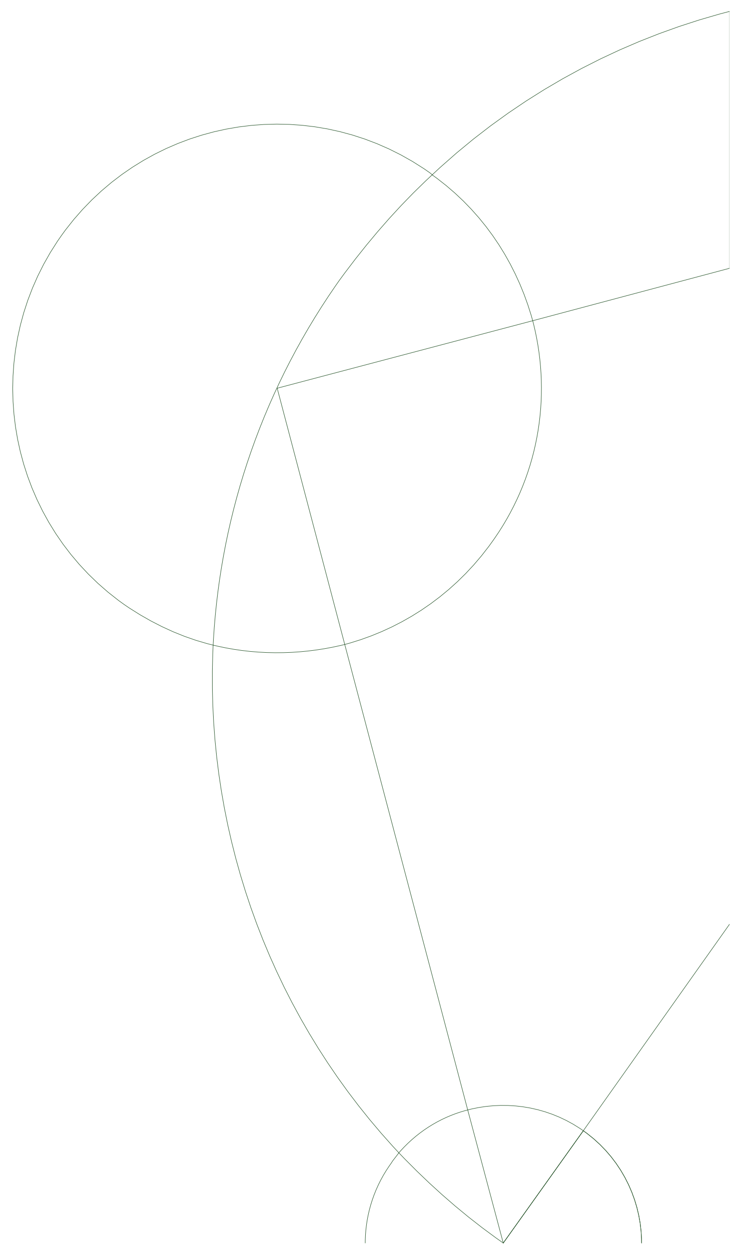


Unfolding Carbon

Algorithmic Generation of Planar Fullerene Precursor Molecules using Intrinsic Geometry

Written by: Frank de Morrée

Supervised by: James Emil Avery



June 2021

Frank de Morrée

Unfolding Carbon

MSc. in Physics, June 2021

Supervised by James Emil Avery

University of Copenhagen

Niels Bohr Institute

Blegdamsvej 17

2100 Copenhagen N

ABSTRACT

In this thesis an algorithm was created that, given a fullerene bond-graph, produces planar precursor-molecules that can fold up through autoassembly to the polyhedral molecule, so that it may contribute to the overarching ambition of synthesising fullerene molecules in the lab. The molecules can be represented by unfoldings of the polyhedral surface which can be generated recursively through combinatorial means, using only the bond-graph as input. The possible number of unfoldings using a single fullerene graph are astronomical. Therefore, the search space was reduced by implementing physical and symmetry constraints as to find the most interesting unfoldings. The resulting algorithm shows both high correctness and modularity, but performance concerns of the current Python implementation will have to be addressed in the future.

Keywords: CARMA; fullerene; precursor; molecule; unfolding; carbon; Python.

SOFTWARE

The software created for this thesis can be found at the link below and contains the following files:

- data
  - C60_data
  - C120_data
  - C140_data
  - C180_data
  - C540_data
    - C540-Ih.mol2 — PyMol file containing 3D model of isomer
    - C540Ih.py — Contains dual_neighbours
- source/python
  - common_functions.py — Contains all functions used in recursive_unfold() function
  - performance.ipynb — Used to generate performance numbers
  - plot_unfoldings.py — Used to generate plots and GIFs
  - plotting_functions.py — Contains all plotting and GIF creation functions
  - recursive_unfold.ipynb — Contains main recursive unfolding algorithm
  - single_unfold.ipynb — Contains unfolding algorithm used for testing

https://github.com/fsd776/thesis-carbon-unfolding

CONTENTS

1 INTRODUCTION 1
2 FULLERENE CONTEXT 4
2.1 A BRIEF HISTORY OF FULLERENES . . . 4
2.2 FULLERENE PRODUCTION . . . 6
2.3 FULLERENE SYNTHESIS . . . 7
2.4 THE CARMA PROJECT . . . 10
2.5 AUTOASSEMBLY SUCCESS . . . 11
3 FULLERENE THEORY 13
3.1 POLYHEDRAL GRAPHS . . . 13
3.2 FULLERENE BOND-GRAPHS . . . 14
3.2.1 Dual graphs . . . 16
3.3 GENERATING BOND-GRAPHS . . . 17
3.4 PRECURSOR-UNFOLDINGS . . . 18
3.4.1 Orientation . . . 24
3.4.2 Adjacency matrices . . . 26
3.5 FULLERENES IN 3D: GEOMETRY . . . 28
3.6 FULLERENE SYMMETRY . . . 30
3.6.1 Abstract groups and point groups . . . 30
3.6.2 Precursor-unfolding symmetry . . . 35
3.7 COMPUTING SYMMETRY . . . 36
4 THE ALGORITHM 38
4.1 PROBLEM STATEMENT . . . 38
4.2 ALGORITHM DESIGN . . . 39
4.2.1 Designing P.1a: Input . . . 39
4.2.2 Designing P.1a: Output . . . 41
4.2.3 Designing P.1a: Unfolding updates . . . 44

4.2.4 *Designing P.1a: Unfolding constraints* . . . 48
4.2.5 *Designing P.1b: Generating all unfoldings* . . . 54
4.2.6 *Designing P.2a: Placing faces* . . . 61
4.2.7 *Designing P.2b: Placing symmetry groups* . . . 67
4.3 IMPLEMENTATION . . . 74
4.3.1 *Implementing P.1a: Input* . . . 74
4.3.2 *Implementing P.1a: Output* . . . 74
4.3.3 *Implementing P.1a: Initialisation* . . . 75
4.3.4 *Implementing P.1b: Generating all unfoldings* . . . 80
4.3.5 *Implementing P.1a: Unfolding constraints* . . . 83
4.3.6 *Implementing P.1a: Triangle constraints* . . . 85
4.3.7 *Implementing P.1a: Face constraints* . . . 86
4.3.8 *Implementing P.1a: Symmetry group constraints* . . . 88
4.3.9 *Implementing P.1a: Hybrid unfoldings* . . . 98
4.3.10 *Implementing P.1a: Updating the unfolding* . . . 100
4.3.11 *Implementing P.1a: Placing triangles* . . . 101
4.3.12 *Implementing P.2a: Placing faces* . . . 103
4.3.13 *Implementing P.2b: Placing symmetry groups* . . . 105
4.3.14 *Plotting unfoldings on the Eisenstein plane* . . . 105
5 RESULTS AND DISCUSSION 108
5.1 RESULTS . . . 108
5.1.1 *Unfoldings* . . . 108
5.1.2 *Performance* . . . 123
5.2 DISCUSSION . . . 124
5.2.1 *Unfoldings* . . . 124
5.2.2 *Performance* . . . 129
5.3 FUTURE WORK . . . 130
5.3.1 *Unfoldings* . . . 130
5.3.2 *Performance* . . . 132
6 CONCLUSION 133
BIBLIOGRAPHY 136
APPENDICES I
A ADDITIONAL UNFOLDINGS I

LIST OF FIGURES

2.1 3D Fullerene model examples 5
2.2 Picture of Richard Buckminster Fuller 6
2.3 The stepwise synthesis of the C_{60}-I_h precursor-molecule 8
2.4 Autoassemby using FVP 9
3.1 Illustration of a simple graph 14
3.2 Planar embeddings of fullerene bond-graphs and their dual 17
3.3 Face spirals 19
3.4 Box molecule unfolding process 22
3.5 All possible box molecule unfolding isomers 23
3.6 Illustration of an arc, triangle, and face 24
3.7 Visualisation of the Eisenstein ring 25
3.8 Box molecule precursor-unfolding 26
3.9 (Sparse) Adjacency matrix for the box precursor-unfolding. 27
3.10 Examples of surfaces with Gaussian curvature 28
3.11 Positive Gaussian curvature 29
3.12 Symmetry elements of hexagonal ring 34
3.13 Dihedral group example 35

3.14 Decision tree for determining the point group of a fullerene 37

4.1 Diagram of general algorithm design 39
4.2 Design dual-neighbour array 40
4.3 Design arc-array 43
4.4 Illustration of considered data structures 43
4.5 Example of possible reversed arcs 45
4.6 Diagram of unfolding method design 46
4.7 Visualisation of the triangle placement cycle 47
4.8 Illustration of updating the workset 48
4.9 Collision example 49
4.10 Speed of dicts. versus lists. 51
4.11 Example of erroneous triangle placement 52
4.12 Diagram of single-unfold process 53
4.13 The squirrel recursion tree 55
4.14 Example of recursive paths 58
4.15 Diagram of general recursion algorithm 59
4.16 Diagram of complete recursive algorithm 60
4.17 Visualisation of determining and placing a face 62
4.18 Possibilities for placing the face with work-arc 63
4.19 Visualisation of excluding duplicate triangles from face placement . 65
4.20 Pentagon cutout example 66

4.21 Visualisation of the symmetry group placement cycle 68

4.22 Visualisation of the symmetry group placement constraints 69

4.23 Pentagon dihedral symmetry 69

4.24 Finding symmetry arcs 73

4.25 Visualisation of the get_pos() function. 84

4.26 Shifting pentagon cutout example 94

4.27 Plotting unfolding example 106

5.1 3D models of the isomers 109

5.2 C_{60}-I_h , C_n unfolding collection 113

5.3 C_{60}-I_h , D_n unfolding collection 114

5.4 C_{120}-D_6 , C_n unfolding collection 115

5.5 C_{120}-D_6 , D_n unfolding collection 116

5.6 C_{540}-I_h , C_n unfolding collection 117

5.7 C_{540}-I_h , C_n unfolding collection 118

5.8 Showcase valid unfoldings 119

5.9 Showcase invalid unfoldings 120

5.10 C_{540}-I_h , root-node 150 variation 121

5.11 C_{540}-I_h , root-node object variation 122

A.1 Selection of unfoldings for the remaining isomers I

1 INTRODUCTION

Fullerenes are a class of molecules formed exclusively from pentagonal and hexagonal rings that form a wide variety of polyhedral cage structures. Polyhedra are the group three-dimensional shapes with flat polygonal faces, and fullerene polyhedra can be built out of anything that forms hexagons and pentagons. The size and three-dimensional shape of the molecule is dependent on the amount of atoms involved, which can be as little as 20 but in theory could go up to an arbitrarily large number of atoms.

Although fullerene structures can be formed from any material, carbon-type fullerenes are of specific interest as they have shown promising properties such as extreme electron mobility and tensile strength (Fischer et al., 1992). The applications for fullerenes are therefore vast, and are apparent today in fields ranging of medicine to electrical engineering (Bakry et al., 2007; Nimibofa et al., 2018). One example given by Bakry et al. (2007) is that because fullerenes can be opened by changing one pentagon into a hexagon, they make compelling candidates for delivery systems on a molecular level. Another example listed in Nimibofa et al. (2018) is that due to the fact that fullerenes have excellent electron accepting capacity, they are widely adopted in photovoltaics in donor-acceptor dyads as well as in artificial photosynthesis electronics.

Despite the clear interest in fullerene molecules, reliable production of most fullerenes remains difficult in practice. Although the smaller fullerenes of less than one-hundred atoms can be readily obtained in kilogram quantities, yields are impure and do not allow for selectivity in the type of fullerenes produced. To understand the scale of the problem, consider that there are $\mathcal{O}(N^9)$ number of possible fullerene configurations for any $N > 20$ carbon atom fullerene. For example, the C_{400} fullerene

has 132,247,999,328 distinct molecular isomers, and current methods are inadequate to cherry-pick the desired ones (Schwerdtfeger et al., 2015, p. 97).

To tackle the selective production of fullerenes, research was done on the chemical rational synthesis of fullerenes. This formation process of a fullerene is comprised of two stages: First, the formation of a precursor molecule which acts as the building block. And second the autoassembly stage, in which the precursor molecule is prompted by an energy impulse such as heat or light to fold into the desired fullerene structure. Using this method, Scott et al. (2002) were able to successfully synthesise C_{60}-I_h. In the study, a method known as *Flash Vacuum Pyrolysis* (FVP) was used to fold the precursor to the stable finalised fullerene molecule. The pyrolysis technique involves intensely and briefly heating the precursor molecule, after which the molecule should assemble to a predetermined structure (Wentrup, 2014). However, the yields of the final product using FPV were as low as 0.1-1.0%, mainly due to the harsh reaction conditions (Scott, 2004; Scott et al., 2002). In subsequent years the method was vastly improved upon, first by Otero et al. (2008) using a cyclodehydrogenated C_{60} precursor on a platinum surface who achieved a near 100% yield at the autoassembly stage; and later on by Kabdulov et al. (2010), where fluorine was placed in key locations on the precursor molecule to help control the folding process.

The promising results of fullerene synthesis so far has opened up a new avenue for fullerene research, seeking to deepen the understanding of the autoassembly process from precursor to final state. In physical terms this is a many body molecular dynamics problem, and although there are simulation packages that can handle the quantum chemistry involved current calculation methods are incredibly slow, as even fullerenes with $N \leq 100$ atoms can take weeks of computation time on relatively large computer clusters (Heuser, 2020).

Enter the CARbon MAnifolds project, CARMA for short, that was set up over the past few years by James Avery at the Niels Bohr Institute [1]. CARMA aims to sidestep

[1] For more information please visit https://www.nbi.dk/~avery/CARMA/index.html

the conventional week-long calculations completely by using intrinsic surface geometries of fullerene structures to determine the molecular properties of the fullerenes. The project is investigating the possibility to calculate the same molecular properties quantum-chemical simulations would using nothing but discrete geometry (Avery, 2021). On the basis of the theoretical framework it should moreover be possible to construct algorithms that can analyse fullerenes orders of magnitude faster than is possible at the moment. In the end, the created methods should aid in finding rational synthesis paths for any desired fullerene structure.

As part of the effort, this thesis aims to create algorithms that, given a fullerene bond graph, produce planar precursor molecules that can fold up through autoassembly to the polyhedral fullerene molecule. The molecules can be represented by unfoldings of the polyhedral surface which can be generated recursively through combinatorial means, using only the bond graph as input. The possible number of unfoldings using a single fullerene graph are astronomical. Therefore, the search space must be reduced by determining and implementing physical and symmetry constraints as to find the most interesting unfoldings. This project thus aims to create fullerene unfolding algorithms with high correctness and modularity, so that it may contribute to the overarching ambition of synthesising fullerene molecules in the lab.

2 FULLERENE CONTEXT

2.1 A BRIEF HISTORY OF FULLERENES

Fullerenes are a class of polyhedral molecules consisting exclusively of rings of atoms that form pentagons and hexagons. For several fullerene structures that consist solely of carbon atoms, a wide range of useful applications in fields ranging from engineering to bio-medicine have been found (Nimibofa et al., 2018). However, even though carbon is the fourth most abundant element in the universe, carbon fullerenes have not been found to be widespread in nature at all. In fact, they are mainly discovered in minute amounts in the outflows of carbon stars in outer space, which is possible due to fullerenes' exceptional thermal stability and photochemical properties (Becker et al., 2000; Woods, 2020). On Earth, discoveries have been limited to fullerenes of the form C_{60}, C_{70}, C_{76} and C_{84}, with C_N implying N carbon atoms. Natural occurring fullerenes were always found in minute quantities, formed either by lightning discharges in the atmosphere, or hidden in soot and prehistoric rock formations (Buseck et al., 1992; Buseck, 2002). Non-carbon fullerene structures are also found in nature: the protein complexes that form the of the HIV virus are pentameric and hexameric as well, for example. This knowledge is in turn being used to develop countermeasures to the virus, as one study by Marchesan et al. (2005) shows how fullerene derivatives can be used to inhibit replication abilities of the virus.

Fullerenes are a particular arrangement, or allotrope, of carbon. More well-known allotropes of carbon are diamond and the single-atom carbon sheets known as graphene. The diamond and graphene atoms are sp^3 and sp^2 hybridised respectively, meaning diamond atoms bond with four of their nearest neighbouring carbon atoms, while graphene atoms does so with the nearest three. Fullerene atoms, like graphene,

also form sp^2 orbital hybrids by bonding with their three nearest neighbours, which is also referred to as being *3-connected*. A variety of carbon fullerenes are shown in Figure 2.1.

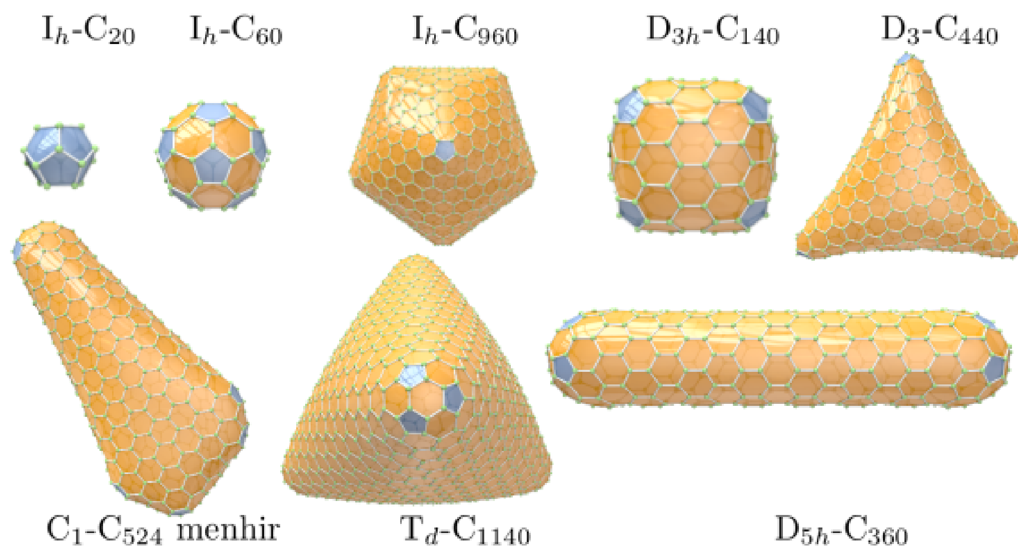


Figure 2.1: Illustration of a number of three dimensional fullerene structures, with the C_{60}-I_h Buckyball as a notable entry in fullerene history.

Early scientific literature on fullerenes is inextricably tied to the *Buckminsterfullerene* C_{60}-I_h. Informally known as the "Buckyball", this highly symmetric icosahedral structure was originally theorised independently by both Õsawa (1970) and Stankevich et al. (1984), later discovered experimentally by Kroto et al. (1985) using laser evaporation techniques on graphite, and first synthesised in larger amounts by Krätschmer et al. (1990). Buckminsterfullerene is named after the American architect Richard Buckminster Fuller, because he was constructing geodesic polyhedral buildings in the late forties as shown in Figure 2.2, because they showed extreme strength compared to their size. Unknown to him at the time, the structure closely resembles the C_{60}-I_h fullerene. Eventually fullerenes became the definition of the class of polyhedral 3-connected carbon structures.

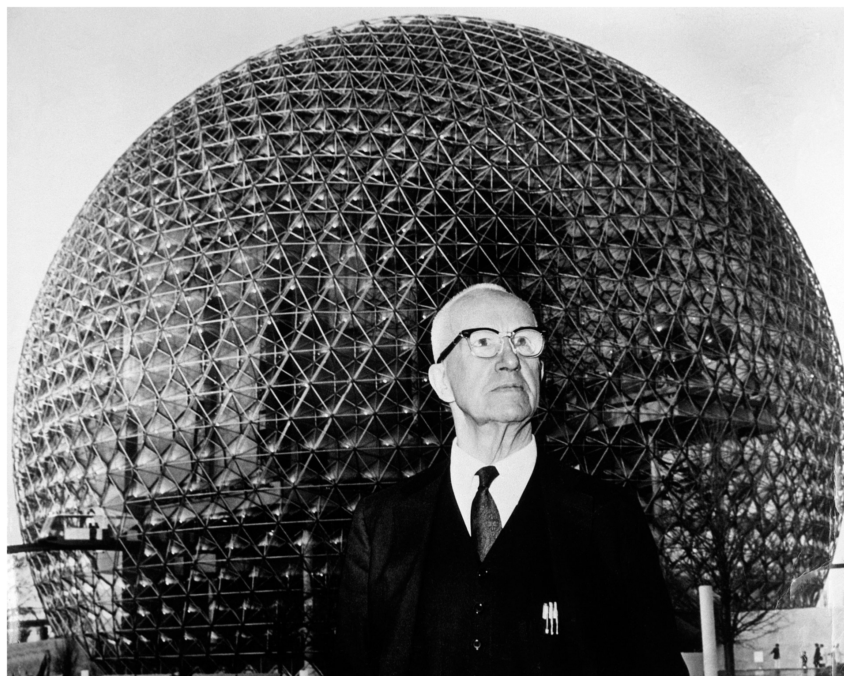


Figure 2.2: The American architect Richard Buckminster Fuller in front of one of his geodesic dome creations. From WIRED (2016).

2.2 FULLERENE PRODUCTION

Given the myriad of applications, there are clear reasons for wanting to produce carbon fullerene molecules. Currently, the fullerenes that can be readily produced in significant quantities are the C_{60}-I_h buckeyball and C_{70}-$D5h$ fullerene, and to a lesser extent C_{76}, C_{78} and C_{84} (Langa & Nierengarten, 2007). The most important method for the production of fullerenes is the Hufmann-Krätschmer method, which involves vaporising two carbon rods in a reactor and extracting the fullerene molecules from the resulting soot. The vaporisation is done by running a high current through the two carbon electrode rods with a diameter of around $6\,\mathrm{mm}$ in a helium filled reactor chamber. With enough current, temperatures of at least $2000\,^{\circ}\mathrm{C}$ form an electrical arc between the electrodes which creates plasma. Over the course of 24 hours, the plasma melts off $100\,\mathrm{g}$ to $200\,\mathrm{g}$ of fullerene-containing soot (Langa & Nierengarten, 2007, p. 3). The resulting carbon mixture can then be purified using column chromatography, resulting in tens of grams of both C_{60} and C_{70}.

Fullerene	No. of isomers
C_{20}	1
C_{60}	1,812
C_{100}	285,914
C_{200}	214,127,742
C_{400}	132,247,999,328

Table 2.1: There are $\mathcal{O}(N^9)$ isomers for each N-atomic fullerne molecule, creating an immense search space.

Clearly, the quantities obtained from a reactor through the arc discharge method are not sufficient for the large scale production of fullerenes. A method that was able to produce larger quantities of fullerenes was discovered as early as 1991 by Howard et al. (1991). Howard and his co-workers proposed to combust benzene in tanks of oxygen deficient environment to generate larger amounts of fullerene soot. After years of development, the combustion method was employed in purpose-built factories, which are each able to produce thousands of tons of fullerenes every year (Chae et al., 2014).

2.3 FULLERENE SYNTHESIS

What is problematic about the macroscopic production methods of fullerenes is that they are neither selective in terms of the different fullerene isomers, nor able to generate higher-order fullerenes that consist upward of a hundred carbon atoms. This is especially salient knowing that the space of fullerene isomers grows with the number of fullerene atoms N as $\mathcal{O}(N^9)$. This means there are already $1,812$ different configurations of the C_{60} fullerene, and $285,914$ for C_{100}, as listed in Table 2.1. In other words, only an incredibly small number of fullerene isomers are currently available to use for the incredible applications found. To gain access to the entire space of fullerene isomers, more direct methods of chemical synthesis are required.

The first successful direct chemical synthesis of a fullerene was done by Scott et al. (2002) who managed to create C_{60}-I_h in isolated quantities, meaning no other

fullerenes were formed in the process. The method starts with the stepwise creation of a so called *precursor-molecule*, which can be best described as the assembly package of a desired *target-molecule*, being the fullerene. For C_{60}-I_h, the precursor-molecule used by the researchers is of chemical formula ($C_{60}H_{27}Cl_3$), whose formation process is shown in Figure 2.3. The figure depicts how the production of three molecular 'arms' are attached to a central hexagonal ring to create the precursor-molecule.

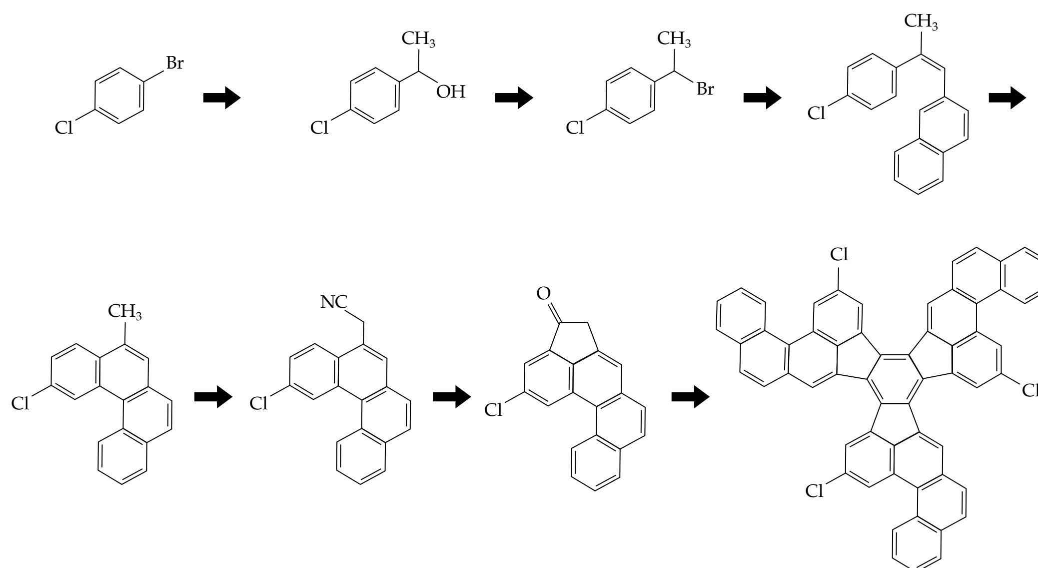


Figure 2.3: The stepwise synthesis of the C_{60}-I_h precursor-molecule by creating molecular arms in parallel. Adapted from Scott (2004, p. 5004).

To autoassemble the precursor to the fullerene, Scott et al. fired a high powered laser pulse at the precursor-molecule under vacuum. By doing so, the precursor is intensely and briefly heated to 1100 °C which initiates a chain reaction that folds the three arms of the precursor together. This technique is known as *Flash Vacuum Pyrolysis*, FVP for short, and is a common technique in the synthesis of organic molecules (Wentrup, 2017). During the folding process, the molecule dehydrogenates (i.e. loses its hydrogen atoms) whilst forming new carbon-carbon bonds. To ensure the correct carbon atoms bond together, three chlorine atoms are placed in strategic locations to guide the process. The result after FVP is the exclusive formation of the C_{60}-I_h fullerene with yields of 0.1-1.0%. The process is visualised in Figure 2.4, where it can be noted that some of the hexa- and pentagonal faces of the 3D molecule

(shown in orange and blue respectively) are not present in the precursor-molecule, but rather are formed during autoassembly.

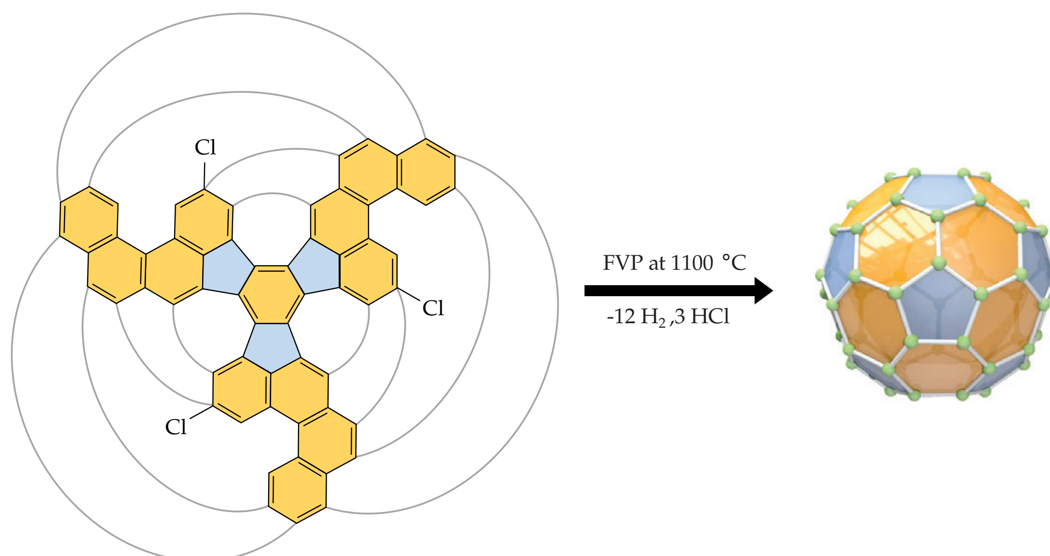


Figure 2.4: Autoassembly of a icosahedral C_{60} precursor-molecule using Flash Vacuum Pyrolysis and chlorine atoms placed in strategic locations. The bonds formed during the process are indicated by the curved lines. Adapted from Scott et al. (2002, p. 1501).

In chemistry, the production of a compound using a sequence of chemical reaction steps is known as *rational synthesis*. The work by Scott et al. (2002) proved that rational synthesis of fullerenes was both possible and selective, albeit with very low yields. Years later, improvements on the method were made by both Otero et al. (2008) and Kabdulov et al. (2010), resulting in yields of nearly 100% of the desired C_{60}-I_h fullerene. Despite the promising results for the rational synthesis of fullerenes, non of the millions of larger fullerene isomers can be synthesised as of yet. The biggest challenge lies in finding the right fullerene isomers that both have valuable properties and are stable during and after autoassembly. What is more, even if suitable candidates for autoassembly would be identified, the accompanying precursor-molecule would also need to be determined. Thus, what is required are methods to sift through the isomer space and determine interesting and viable target- and precursor fullerene molecules for rational synthesis.

The evaluation of fullerene isomers for their potential in rational synthesis is done on the basis of their molecular properties. Using quantum-chemical simulations, qualities such as thermal conductivity, magnetisability and compressive strength can be determined by computing energy or waveform responses to system perturbations (Jensen, 2017). Although the use of such computational methods yields highly accurate results, calculations can take multiple weeks of computation time for a single isomer, even when done on moderately sized computer clusters (Heuser, 2020). Therefore, faster alternative computational methods are needed to identify stable synthesis candidate isomers for every possible N-atom fullerene isomer space.

2.4 THE CARMA PROJECT

To find a way to avoid the computationally expensive quantum-chemical simulations, the CARbon MAnifolds (CARMA) project was established by James Avery at the Niels Bohr Institute in Copenhagen over the past couple of years. The project aims to investigate the possibility of calculating the molecular properties of fullerene structures using nothing but their intrinsic surface geometries derived from their bond-graphs, which will be elaborated upon in the following sections. Moreover, it seeks to find out if the same formalism can be used to find pathways of rational synthesis for fullerene isomers that are found to be of interest.

The search for rational synthesis pathways is referred to as Task R, which is short for *Computing Precursors and Recipes for Rational Synthesis of Fullerenes*. Task R is subdivided in tasks G, A and U, with the overarching goal to build software that aids in finding plausible precursor-molecules for any given fullerene isomer. Task G is titled *Generating Precursor Molecules for Autoassembly*, and is the subject of this thesis. The primary objective of the task is to produce suitable precursor-molecule candidates using nothing but the bond-graph that will autoassemble into the polyhedral fullerene structure by means of Flash Vacuum Pyrolysis. Meanwhile, tasks U and A are concerned with understanding, modelling, and simulating the fullerene autoassembly process.

It should be noted that the task of this project comes into play after other algorithms from other CARMA tasks have already identified fullerene isomers that are of interest to rationally synthesise. Therefore, the intention of this project is to generate precursor-molecule configurations that are likely to successfully autoassemble. After a list of precursor-molecule candidates is generated, experimental chemists can take over and attempt to synthesise and autoassemble the found precursors to the intended fullerene molecule.

2.5 AUTOASSEMBLY SUCCESS

To be able to generate precursor-molecules that are likely to autoassemble, it is important to consider what influences the success rate of the autoassembly process. One lesson can be learned from the precursor-molecule from Scott et al. (2002), which is a *planar* molecule, meaning it can be put on a flat surface without having to distort the molecule by breaking any bonds. Planar precursors provide a number of benefits for the folding process. For one, they are unlikely to form the wrong target molecules spontaneously, making them reliable for autoassembly. Moreover, planar molecules are simple to synthesise in a systematic way. For example, the three molecular 'arms' of the precursor from Scott et al. (2002) that connect to the central hexagon can be synthesised in parallel, which is convenient for its production. Lastly, planar molecules are also simpler to work with computationally and mathematically, since the description of the molecule can be limited to two dimensions instead of three. That is not to say that the direct synthesis of non-planar (precursor) molecules is impossible or unfeasible. In fact, research is being conducted on the direct synthesis of curved (precursor) molecules as well, by Mojica et al. (2013) and Majewski and Stępień (2018) for example. Notwithstanding, planar-precursors are chosen as the direction of exploration in CARMA Task R and this project.

Another indicator for the stability of the fullerene is the way in which its pentagon rings are arranged on the polyhedral surface. Specifically, Albertazzi et al. (1999) have shown that the optimal geometries of fullerenes have a minimal adjacency of their pentagonal faces. In other words, they found the most stable chemical

geometry, and therefore the lowest total energy of the molecule, is achieved when pentagons do not share a bond with any other pentagon. This happens to be the case for the C_{60} buckeyball from Figure 2.4 as well. Moreover, the researchers found that the total energy (and therefore instability) increases linearly with the number of adjacent pentagons (Albertazzi et al., 1999). This phenomenon is called the *Isolated Pentagon Rule* (IPR), and makes it of interest to minimise the number of neighbouring pentagons in fullerene precursor-molecules.

It was already mentioned that it is helpful to be able to construct parts of precursor-molecules in parallel, for example the three arms from the precursor by (Scott et al., 2002). Another reason for wanting this is to increase the overall yield of precursor production. To understand this, consider the precursor synthesis process from Figure 2.3. In every step, it is impossible to successfully create 100% of the desired partial molecule. Consequently, there is a multiplicative loss of the total yield that grows with each additional synthesis step that is required. As such, it is more efficient to minimise the number of compounds, and thereby synthesis steps, that are needed to create the precursor-molecule.

Hence, it is beneficial for precursor-molecules to be maximally symmetric. Of course, the separate parts of the precursor should always be connected together before autoassembly through a method such as FVP. The reason for this is that FVP requires a stationary and stable precursor-molecule that can sit around until it is activated to fold by means of the intense heat pulse. If instead the precursor would consist of a number of separate molecular compounds, it is highly likely they would combine together in unpredictable ways. After all, there is no way of knowing whether the compounds are arranged in a way that guarantees autoassembly to the desired target fullerene. In summary, generated precursor-molecules should ideally be planar, maximally symmetric and adhere to the Isolated Pentagon Rule, to provide the highest chances for autoassembly success.

3 FULLERENE THEORY

In the CARMA project, it was found that the bond-graph of the fullerene contains all the required information to find interesting isomers for synthesis, as well as possibly find paths for rational synthesis. In this section, the graph theory of fullerenes is discussed to identify what the available information is for the development of an algorithm that can generate planar precursor-molecules.

3.1 POLYHEDRAL GRAPHS

In discrete mathematics, a *graph* is a pair $G = (\mathcal{V}, \mathcal{E})$ which describes a set of vertices (also called nodes) $\mathcal{V}$ that are connected by a set of edges $\mathcal{E}$, a simple example of which is shown in Figure 3.1. Graphs describe relations between objects by means of their connections with each other, represented as unordered and unique pairs of vertices. If a graph can be drawn in a two-dimensional plane without having edges that cross, it is called a *planar graph*. Moreover, any drawing of a graph in space is called an *embedding* of the graph. Put differently, a graph is embedded when each of its vertices is assigned a coordinate in two-dimensional real space as a map $\mathcal{V} \to \mathbb{R}^2$. By this logic, a *planar embedding* is a drawing of a planar graph in the two-dimensional plane without any crossing edges.

In general, the embedding of a planar graph is not unique, but there is an exception to this rule. If a graph does not break into disconnected components whenever fewer than three vertices are removed, it is a *three-connected* graph and its embedding is essentially unique (Whitney, 1932). The graph shown in Figure 3.1 is an example of a one-connected graph, because the removal of vertex B results in a disconnected graph with two components.

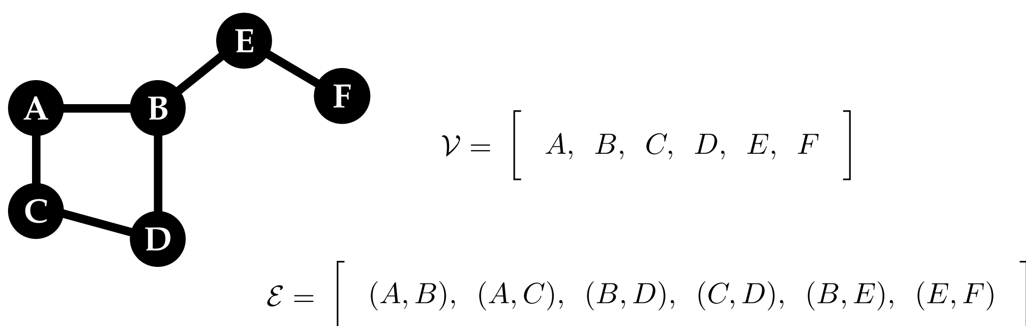


Figure 3.1: Illustration of a simple graph with six vertices and seven edges.

In planar embeddings, regions bound by a set of vertices and edges are known as *faces*, and the complete set of faces is labelled as $\mathcal{F}$. An example of a face is (A, B, D, C) in the graph from Figure 3.1. Every planar embedding is also said to have an *outer face* that is of infinite size and surrounds the graph. Given that three-connected planar graphs are unique, they have a well-defined set of faces. Therefore, a three-connected planar graph can be unambiguously referred to as $G = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ for convenience. An interesting property of planar graphs is that they can be embedded onto the surface of a three-dimensional sphere without crossing edges. What is more, if the embedded graph is three-connected it actually describes a three-dimensional polyhedron. In fact, this is the case for all three-connected planar graphs, which are therefore also known as *polyhedral graphs*.

Another property of graphs is the number of neighbours each of its vertices v has. This number is known as the *degree* of the vertex, and can be written as deg(v) or k. If each of the graphs vertices have the same degree k, the graph is called k*-valent* or k*-regular*. In the specific case where $k = 3$ a graph is called *cubic*, meaning each of its vertices is of deg(3).

3.2 FULLERENE BOND-GRAPHS

Fullerenes can also be described using graphs, specifically the graph that describes their bond-structure, which is called the *bond-graph*. For the fullerene bond-graph, each carbon atom is a vertex, each atomic bond an edge and each face either a

pentagon or hexagon. Fullerene bond-graphs are both cubic, planar, and three-connected, which makes them cubic polyhedral graphs. Because of these neat properties, fullerene graphs have a lot of elegant mathematics surrounding them that can be used to derive many properties about fullerene topology, spatial shape, surface, and even indicators of their chemical behaviour (Schwerdtfeger et al., 2015). One of the interesting properties of fullerene structures is that they always exactly have twelve pentagonal faces, no more and no less, which comes as a direct consequence of the nature of the bond-graph. This principle is called the '12 Pentagon Theorem', the proof of which starts with *Euler's polyhedron formula*:

$$N - E + F = 2 \tag{3.1}$$

Here, $N = |\mathcal{V}|$ is the total number of vertices and is called the *order* of the graph. Similarly, $E = |\mathcal{E}|$ represents the number of edges, and $F = |\mathcal{F}|$ the number of faces. Furthermore, the hand-shaking lemma from graph theory states:

$$\sum_{i=1}^{N} \deg(v_i) = 2E \tag{3.2}$$

Combined with the fact that $\deg(v_i) = 3$ for all vertices of a fullerene, one gets:

$$E = \frac{3}{2}N, F = \frac{1}{2}N + 2 = \frac{1}{3}E + 2, \text{and} E = 3F - 6 \tag{3.3}$$

In addition, the total number of faces of a fullerene is the sum of all its pentagons and hexagons as $F = F_5 + F_6$. From this, together with Equation (3.3) it can be derived that:

$$E = 3F_5 + 3F_6 - 6 \text{and} N = 2F_5 + 2F_6 - 4 \tag{3.4}$$

Since every pentagon and hexagon have five and six edges respectively, another way to define the total number of edges is to use the number of faces and the number of edges per face as:

$$E = \frac{5}{2}F_5 + \frac{6}{2}F_6 \tag{3.5}$$

The division by two arises due to the face that every edge (bond) is shared between two fullerene faces, and should therefore not be counted twice. Using eq. (3.4) and eq. (3.5) together yields:

$$\begin{aligned} 3F_5 + 3F_6 - 6 &= \frac{5}{2}F_5 + \frac{6}{2}F_6 \\ (3 - \frac{5}{2})F_5 + (3-3)F_6 - 6 &= 0 \\ \frac{1}{2}F_5 &= 6 \\ F_5 &= 12 \end{aligned}$$

which is proof for the 12 Pentagon Theorem. Similarly, by also using eq. (3.1), the number of hexagons is equal to:

$$F_6 = \frac{(N-20)}{2}(N \geq 20) \tag{3.6}$$

This makes C_{20} consisting of twelve pentagons the smallest possible fullerene, and moreover gives the general formula for the number of atoms N as C_{20+2F_6}.

3.2.1 *Dual graphs*

To be able to draw, generate, and transform the fullerene bond-graph, it is convenient to work with a type of graph representation known as the *dual graph* (Schwerdtfeger et al., 2015). In the dual graph G^* of a planar connected graph G, there is a vertex in G^* for each face of G and an edge joining every pair of neighbouring faces of G together. Mathematically, the operation from the cubic bond-graph to the triangular dual graph is an involution, meaning $(G*)^* = G$. For the fullerene bond-graph, The dual graph of a cubic polyhedral fullerene bond-graph is a triangulation that has a vertex at the centre of each pentagonal and hexagonal face. In Figure 3.2, example planar embeddings of fullerene bond-graphs, together with their dual graphs and three-dimensional dual embeddings are shown. It can be observed that each pentagon and hexagon give rise to dual vertices of degree 5 and 6 respectively, with each dual edge connecting the faces perpendicular to the carbon bonds that connect the atoms.

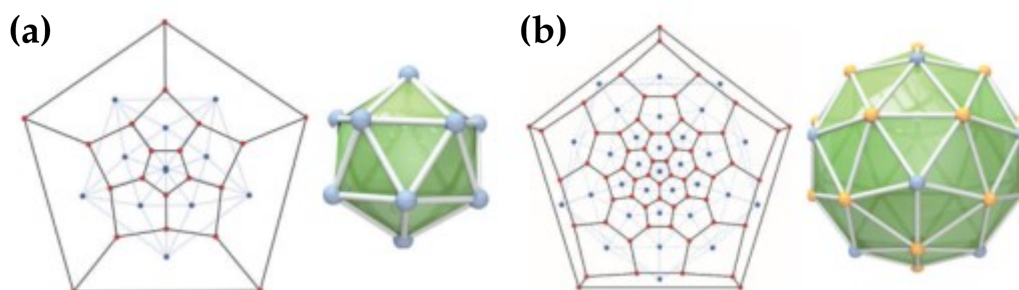


Figure 3.2: Planar embeddings of fullerene bond-graphs and their dual (in blue), together with 3D embeddings of the dual graphs for both **(a)** C_{20}-I_h and **(b)** C_{60}-I_h.

Drawing planar embeddings of fullerene graphs as in Figure 3.2 can be imagined as squishing the 3D polyhedral embedding (either cubic or dual) to the 2D plane without breaking any bonds. There are many possible ways to do so, the choice of which depends on the type of information that needs to be visualised, such as the symmetry of the molecule or the shape of the surface. For more information on the possible drawing methods and the one used in Figure 3.2, see Schwerdtfeger et al. (2015, p. 100).

3.3 GENERATING BOND-GRAPHS

To be able to explore the large isomer space of fullerenes, it is important to have a reliable yet exhaustive method of generating their bond-graphs. Developments in efficient graph generation started with work done by Brinkmann et al. (1998), who used a process of transforming an initial small fullerene graph into new and larger ones. The process involved taking the C_{24} bond-graph as a start and adding faces to it step by step, by which they were able to generate nearly every graph up to C_{200}. Further improvements on the method were done by Hasheminezhad et al. (2008), who defined a set of growth operations that were able to produce every possible graph starting from C_{20} systematically. In the following years, Brinkmann et al. (2012) took the set of growth operations to create an algorithm able to generate every fullerene isomer, which they used to generate a complete database up to C_{400}.

To find fullerene isomers that would be interesting to synthesise, it is less relevant to generate the incredibly large isomer lists using the methods from Brinkmann et al.

(2012). Instead, a more selective algorithm is needed that is able to generate fullerene graphs of isomers with desirable properties. This is one of the completed goals of the CARMA project, which resulted in an algorithm that is part of the *Fullerene* software package created to bundle parts of the research from the CARMA project. More information on the software can be found by reading Schwerdtfeger et al. (2013), and the development version of Program Fullerene can be found here: link.

The particular algorithm that is able to construct the bond-graph of specific isomers of any size (e.g. $C_{100,000}$ in a matter of seconds) uses a generalised spiral construction method outlined in Wirz et al. (2018). In short, the method unwinds the fullerene faces 'like an orange peel' on the basis of the pentagon positions of the fullerene. This results in a spiral representation of the fullerene molecule, which can be represented as a list of so called *face spiral pentagon indices* (FSPI), denoted as $\{S_n|n = 1, ..., 12\}$. The spiral is a string of length equal to the number of faces F, consisting of 12 fives and $F - 12$ sixes, for the pentagons and hexagons. There are $6N$ spirals per fullerene bond-graph, because there are $6N$ possible ways to start the spiral. For each possible fullerene isomer however, there is a unique FSPI, also known as the *canonical* FSPI, when the indices are lexicographically ordered. Two examples are shown in Figure 3.3, where the canonical spirals and FSPI of both C_{60}-I_h and C_{100}-T_d are shown. Every spiral produced with the generalised algorithm contains all the required information of a fullerene bond-graph in only twelve numbers, and can be used to construct the (dual) fullerene bond-graph in $\mathcal{O}(N)$ time.

3.4 PRECURSOR-UNFOLDINGS

At this point, the question becomes how to construct a representation of a precursor-molecule from a generated bond-graph of a particular fullerene isomer. To provide an answer, consider that a fullerene is formed by folding a precursor-molecule, making the precursor simply an unfolded version of the fullerene. Unfolding a fullerene molecule can be imagined as cutting carbon-carbon bonds with a pair of scissors until the resulting molecule can be laid down onto a plane, as if the fullerene was made of intricately folded paper. In other words, if the surface of a fullerene can

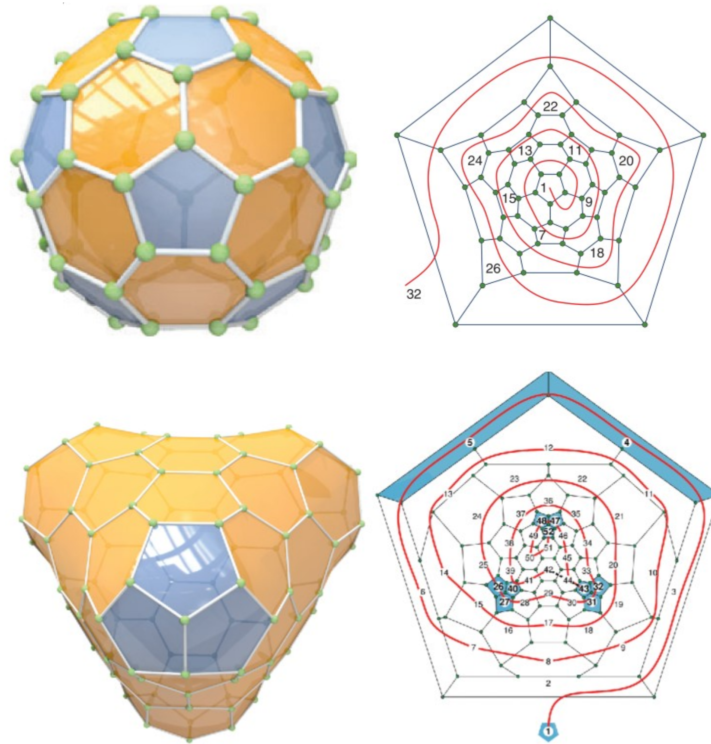


Figure 3.3: The face spirals of the C_{60}-I_h with FSPI $\{1, 7, 9, 11, 13, 15, 18, 20, 22, 24, 26, 32\}$ (top), and C_{100}-T_d with FSPI $\{1, 4, 5, 26, 27, 31, 32, 40, 43, 47, 48, 52\}$ (bottom) fullerenes. From Schwerdtfeger et al. (2015) and Wirz et al. (2018).

be embedded onto a plane such that each of the atoms are placed once, the resulting unfolding represents its precursor-molecule. Earlier work in the CARMA project discovered that it is algebraically heavy handed to embed the cubic polyhedral surface to a plane (Wirz, 2015, p. 20). The dual graph on the other hand, can be conveniently unfolded to a plane of equilateral triangles known as the *Eisenstein plane*. The grid points that form the triangular lattice of the Eisenstein plane are formed by the countable infinite set of *Eisenstein integers*:

$$(a, b) = a + b\omega \quad \text{with} \quad \omega = e^{i\frac{2\pi}{6}} \tag{3.7}$$

The coordinates in the Eisenstein plane are labelled as (i, j), which relate to Cartesian coordinates (x, y) as:

$$\begin{aligned}
(1_i, 0_j) &= (1_x, 0_y) \\
(0_i, 1_j) &= (\cos(2\pi/6),\ \sin(2\pi/6)) \\
\Rightarrow (x, y) &= (i + j \cdot \cos(2\pi/6),\ j \cdot \sin(2\pi/6))
\end{aligned} \tag{3.8}$$

Using the fact that $\omega^2 = \omega - 1$, multiplication of Eisenstein integers can be used to translate the coordinate $(1, 0)$ into any of the other grid points:

$$(a, b) \cdot (c, d) = ac + (ad + bc)\,\omega + bd\omega^2 = (ac - bd, bc + (a + b)\,d) \tag{3.9}$$

It is important to understand the distinction between an unfolding of the cubic polyhedral surface and that of the dual triangular surface. The first is an embedding of all the polyhedral faces that make up the molecule, and can therefore be referred to as a *polyhedral unfolding*, whilst the latter is an embedding of all the dual surface triangles. Since every triangle is only embedded once, and represents a cubic node which is an atom, the unfolding is one of the precursor molecule: a *precursor-unfolding*.

To better understand the distinction, the unfoldings are visualised for a fictional box molecule in Figure 3.4. For the polyhedral-unfolding, bonds are cut laterally, thereby splitting the edges in two. Cubic nodes (atoms) need to be duplicated to keep the faces (squares) intact. The result is that the periphery of the unfolding consists of the split edges and duplicated nodes. To create a precursor-unfolding, the edges of the dual graph are cut laterally. This results in the cubic graph edges (i.e. atomic bonds) to be broken transversely, leaving half-bonds sticking out of the triangles that lie on the periphery. Again, the dual nodes need to be duplicated to ensure the triangles are embedded intact. However, because the atoms lie at the centroids of the triangles, the atoms are unique in the dual representation.

Another important takeaway from the figure is that unfoldings are not the same as planar embeddings of the bond-graph, because nodes and edges are not repeated

in the planar embedding. Moreover, whereas the planar embedding of the cubic bond-graph is unique, there are numerous ways to create unfoldings. All possible isometries for the box molecule unfoldings are shown in Figure 3.5, illustrating the fact that various precursor-molecule isomers that can fold up to the same target-molecule isomer.

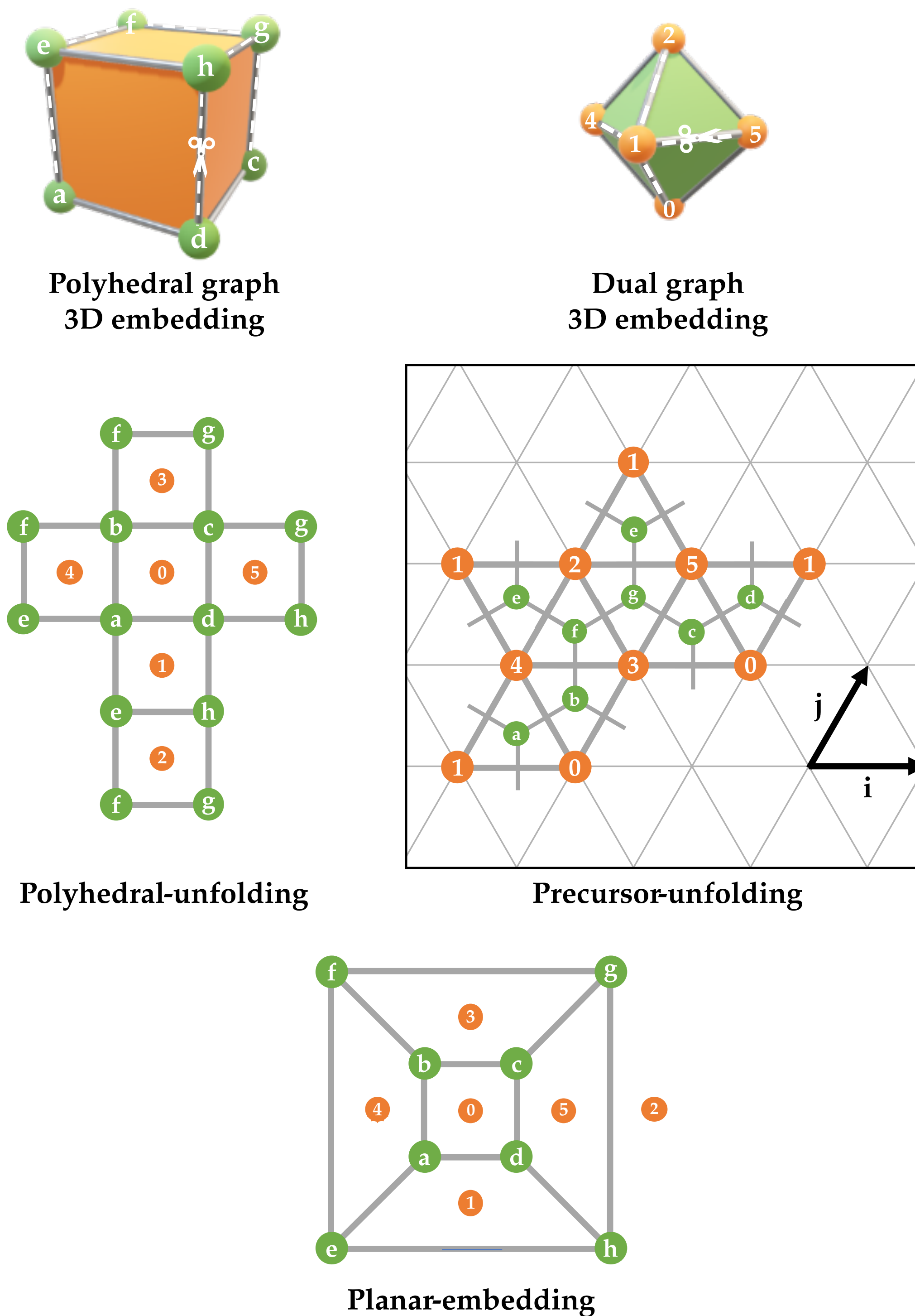


Figure 3.4: A polyhedral box molecule is unfolded in its cubic (left) and dual (right) graph form by cutting edges laterally. In contrast to a planar-embedding, this requires duplication of peripheral edges and nodes.

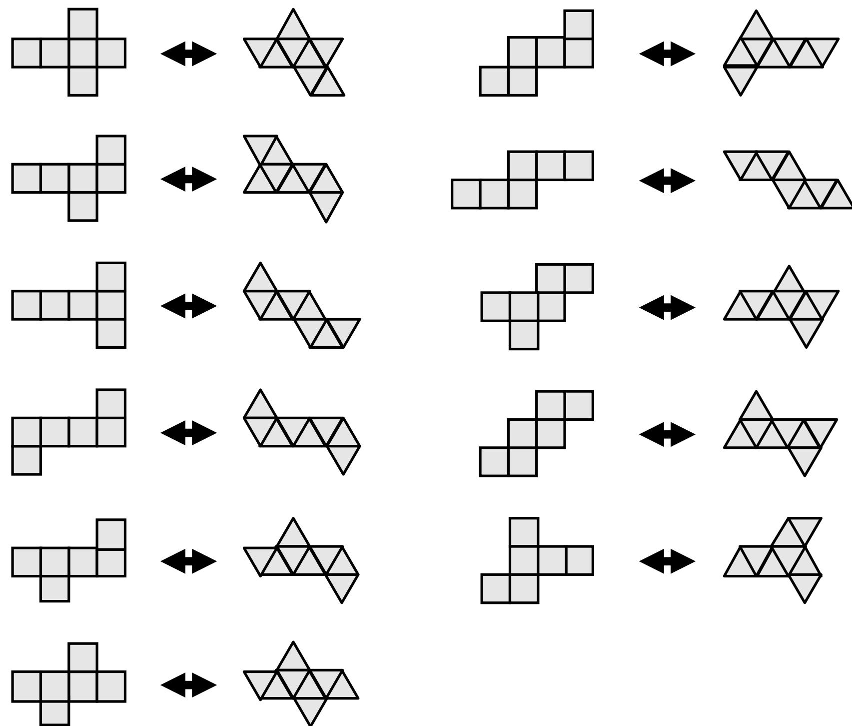


Figure 3.5: All possible box isometric unfoldings of both the cubic (square) and dual (triangular) surfaces.

3.4.1 Orientation

To make mathematics with the unfoldings more convenient, it is useful to define an *orientation* of the graph. In such a *directed-graph*, each edge becomes bi-directional, and each uni-directional split edge is called an *arc*. The arcs are ordered pairs of nodes, with a *source* node and *target* node as the start and end point of the arc respectively. To set the direction of the arcs, the graph is chosen to be oriented either clockwise or counterclockwise. For this and other CARMA projects, clockwise was chosen as the default direction. As a result, each dual triangle becomes a unique set of arcs that are embedded in clockwise direction on the Eisenstein plane. Moreover, the periphery of the precursor-unfolding follows a clockwise path on the plane, as exemplified for a pentagon and hexagon in Figure 3.6. Moreover, it can be seen in the figure how adjacent triangles share a bi-directional edge, even though the two constituent arcs are only part of one unique triangle. For example, edge $(u, w) = (w, u)$ in the pentagon is part of triangles (u, v, w) and (u, w, q), but arc (w, u) is only part of the first, and (u, w) only part of the latter triangle.

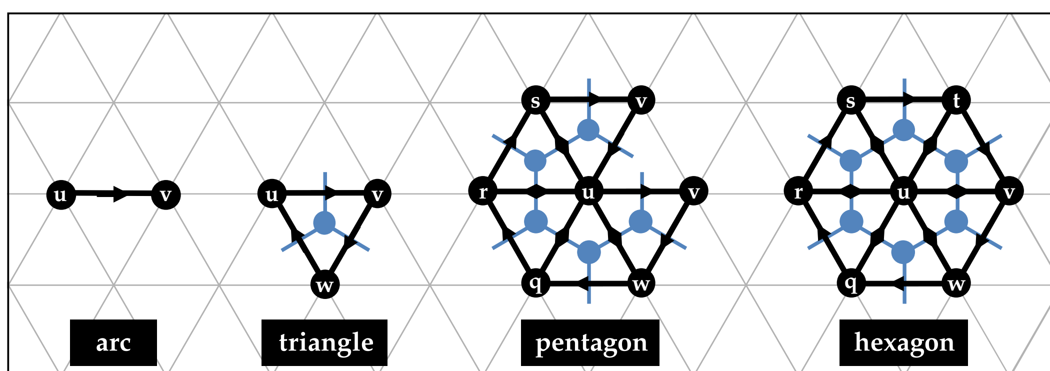


Figure 3.6: An unfolding in the dual representation if formed by arcs that form triangles defining a single carbon atom with three bonds (blue nodes). In turn, triangles form pentagons and hexagons that create fullerene unfoldings.

The possible orientations an arc can have in the Eisenstein plane are given by the *Eisenstein ring*, which is shown in Figure 3.7. The ring is the hexagon formed by the vectors pointing in each of the unitary combinations of (i, j). Any given arc orientation $(i_u, j_u), (i_v, j_v)$ can be rotated into the other orientations by means of one or more Eisenstein multiplications of the target node coordinates in either clockwise

or counterclockwise direction using Equation 3.9. The corresponding Eisenstein integers are $(1, -1)$ for clockwise and $(1, 0)$ for counterclockwise rotations in the ring. In the figure, a single clockwise rotation of the target node v is shown as

$$(-1, 0) \cdot (1, -1) = (-1, 1)$$

The result can be used to find the third node w of the clockwise oriented triangle that is defined by the arc (u, v), which will be discussed further in the following chapter.

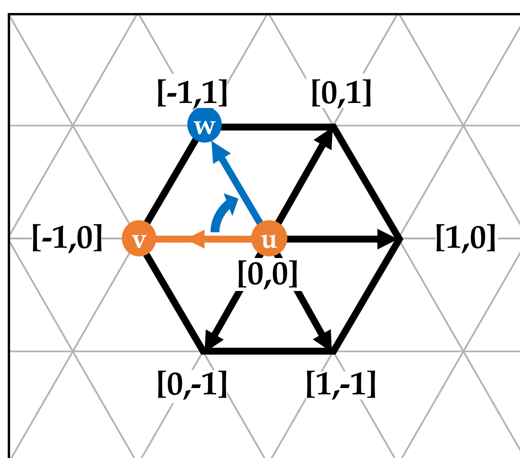


Figure 3.7: Eisenstein multiplication of the end node $v : (-1, 0)$ path in clockwise direction $(1, -1)$ on the Eisenstein ring results in the new path with end node $w : (-1, 1)$.

A benefit of setting an orientation for the graph is that it makes it possible to identify triangles and cubic faces by only one of its arcs. This is important because a valid precursor-unfolding does not have duplicate atoms, meaning each triangle should be identified and only embedded on the Eisenstein plane once. In the pentagon and hexagon of Figure 3.6, arc (u, v) defines the triangle (u, v, w), while arc (u, r) defines triangle (u, r, s), for example. The same goes for the whole cubic faces however, as arc (u, v) is only part of the face that surrounds the dual node u. Even if the faces were part of a larger molecule, they can be uniquely identified as the face with node u at its center, that forms arcs with its neighbours (v, w, q, r, s, t), which in turn define the five or six triangles that form the pentagon and hexagon faces.

To gain better understanding of the directed-graph unfolding, a depiction of the box precursor-unfolding with clockwise orientation is shown in Figure 3.8. In the figure, arcs that share a node and are oppositely directed are part of the same edge, and will therefore align during autoassembly. For example, arc $(3, 0)$ will fold towards $(0, 3)$ to form a carbon bond between atoms b and c. In the same way, duplicate dual nodes merge together during the folding process, recreating the 3D embedding that was shown at the top right of Figure 3.4. It should be noted that singular, unconnected arcs have no physical meaning in the unfolding, as only complete triangles represent atoms. Hence, even though arcs uniquely define triangles, they should always be embedded as part of full triangles on the Eisenstein plane.

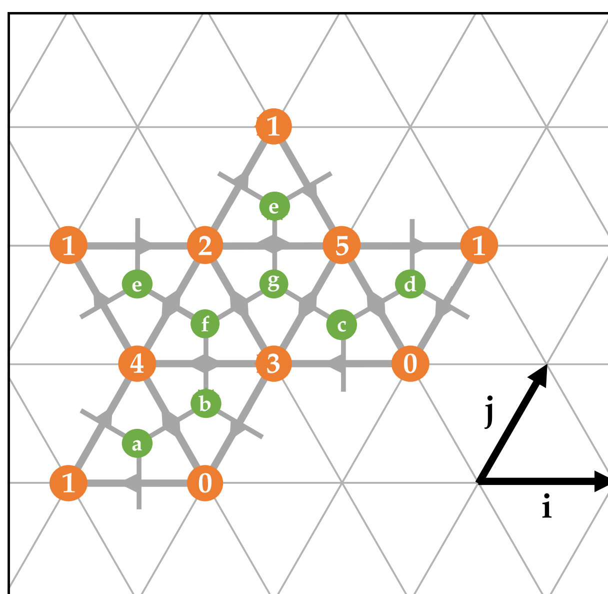


Figure 3.8: A box molecule precursor-unfolding set in clockwise orientation. Oppositely directed arcs realign during autoassembly.

3.4.2 *Adjacency matrices*

To construct an algorithm that creates precursor-unfoldings, a useful way to represent the bond-graph as input data is needed; which moreover needs to be unique for each fullerene isomer. Fortunately, a representation known as the *sparse adjacency matrix* exactly meets the requirements. In general, an *adjacency matrix* $A_{i,j}$ is a matrix with entries equal to 1 if nodes i and j are connected and 0 otherwise. For the dual

bond-graph G^* the result is an $n \times n$ matrix with n the number of dual-graph nodes. However, this matrix does not give information on the orientation of the graph and is difficult to read. To make the matrix clearer, it can be made *sparse* by taking the indices of the connected nodes that have a value $A_{ij} = 1$ and storing them in a new matrix in clockwise order.

Both the regular and sparse adjacency matrix for the box precursor-unfolding of Figure 3.8 are shown in Figure 3.9. It can be seen in the figure that each row stores the neighbours of the node with the same number as the row index, leading to row 0 of the sparse matrix to have entries $[1, 4, 3, 5]$. In general, the number of columns in the sparse matrix is equal to the number of nodes that create a cubic face of the molecule, which for the square faces of the box molecule is equal to four. It should be noted that there is no meaningful way to define a 'starting neighbour' of a face, i.e. $[1, 4, 3, 5] = [5, 1, 4, 3] = [3, 5, 1, 4] = [4, 3, 5, 1]$ since the clockwise order is preserved.

$$\underbrace{\begin{bmatrix} 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix}}_{\text{Adjacency matrix}} \quad \underbrace{\begin{bmatrix} 1 & 4 & 3 & 5 \\ 0 & 4 & 2 & 5 \\ 1 & 5 & 3 & 4 \\ 0 & 4 & 2 & 5 \\ 0 & 1 & 2 & 3 \\ 0 & 3 & 2 & 1 \end{bmatrix}}_{\text{Sparse adjacency matrix}}$$

Figure 3.9: The (sparse) adjacency matrix A_{ij} for the box precursor-unfolding from Figure 3.8, representing the dual-graph connectivity information.

3.5 FULLERENES IN 3D: GEOMETRY

To better understand the connection between the precursor-unfolding and three-dimensional target-isomer it autoassembles to, it is useful to discuss fullerene geometry. Fullerenes are hollow polyhedral cages that can be described conveniently by a quantity known as *Gaussian curvature*, denoted as K. Gaussian curvature is an intrinsic property of a given surface and is therefore independent of how the surface is embedded in space. The three cases of curvature are shown in Figure 3.10, as K can be either negative, zero, or positive at any given point on the surface. If the Gaussian curvature is negative in a point, it gives a saddle point and the surface is shaped like a pringle. If instead the surface has zero curvature everywhere, it can be compared to a sheet of paper that can lie flat in a plane. Lastly, a surface that has positive Gaussian curvature bends uniformly in all possible directions, giving rise to a spherical surface. On a closed surface with constant Gaussian curvature, the three domains of K would yield the geometry of a hyperboloid, cylinder and sphere respectively.

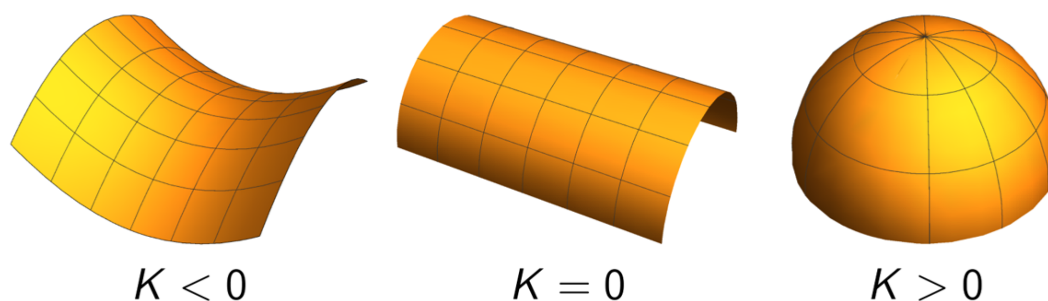


Figure 3.10: Examples of surfaces with negative ($K < 0$), zero ($K = 0$), and positive ($K > 0$) Gaussian curvature around a point. From Avery (2020, p. 10).

Surfaces with negative Gaussian curvature cannot be embedded on a plane because if one would try to squish the pringle-like surface, there would not be enough space on the plane without creating folds and allowing parts to overlap. Meanwhile, zero curvature surfaces can be easily embedded on a plane since they are a flat sheet. Finally, a positive curvature surface can be unwrapped, but only when cuts are made on the surface. For fullerenes, the surfaces are closed and have non-negative

Gaussian curvature everywhere. In other words, at any point on the fullerene the surface is either flat or curving like a sphere. Consequently, the fullerene surface with $K > 0$ can only be embedded on a plane when cuts are made. Specifically, for each of the pentagons one of the bond needs to be cut, while the hexagons can remain intact when unfolding them onto the plane. The cut results in a triangular wedge cutout as shown for a partial molecule in Figure 3.11. The cutout has an angle of $2\pi/6$ as part of the full 2π circle, drawn in grey in the figure. By the Bertrand-Diquet-Puiseux theorem, the cutout angle is also the Gaussian curvature that is induced by the pentagon after the split arcs are glued back together, as shown on the right in the figure. Therefore, after all twelve pentagons of a fullerene are folded together, the total Gaussian curvature equates to 4π. This is the same curvature as a sphere, thereby confirming that the fullerene surface is closed at this point. Consequently, the positions of the pentagons in the precursor-unfolding determine how the surface of the fullerene bends and how strongly, which in turn gives rise to the natural shape of the fullerene.

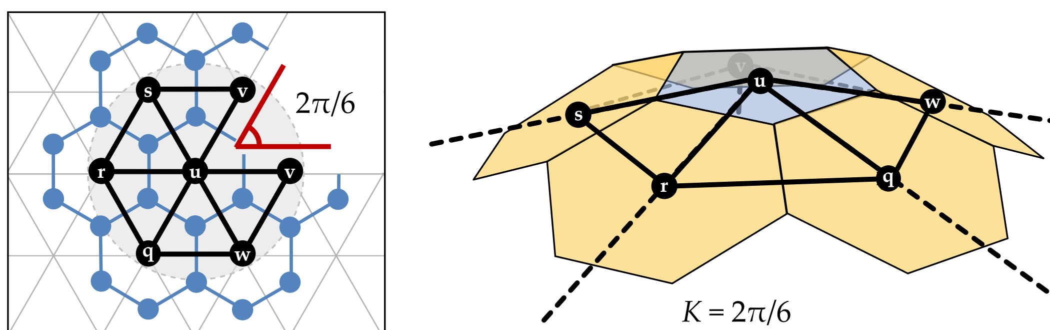


Figure 3.11: Left: A pentagon is embedded as part of a molecule on the Eisenstein plane by cutting out a triangular wedge of angle $2\pi/6$. **Right**: After folding, the pentagon forms an infinite cone that induces a Gaussian curvature of $K = 2\pi/6$ with the surrounding hexagons.

3.6 FULLERENE SYMMETRY

As discussed in section 2.5, creating maximally symmetric precursor-molecules is advantageous because symmetric parts can be produced in parallel. To this end, an understanding of the symmetry of fullerenes and their unfoldings is required.

3.6.1 Abstract groups and point groups

The symmetry of an object is given by a collection of transformations, called *symmetry operations*, under which the object is invariant, meaning the object before and after the operation are indistinguishable from one another. Because mathematical objects do not have to exist in Euclidean (3D) space, the transformations do not necessarily have to take the geometry of the object into account either. Hence, the collection of symmetry operations is referred to as the *abstract group* of the object, denoted $\mathcal{G}$. For a fullerene graph, the abstract group contains all the ways in which the labels of the atoms can be shuffled under which the bond-graph is invariant.

Of course, fullerene molecules do exist in three-dimensional space. If the geometry of the object is taken into account, the symmetry operations are limited to the geometric transformations, which for a molecule are translation, rotation, reflection and inversion. These transformations preserve the angles and distances between all points (atoms and bonds) of the object (molecule), and are therefore called *isometries*. If translation is excluded by fixing the object around a point in space such as the origin, the group containing all the isometries is called the *point group* of the object. The isometric symmetry operations in the point group are done about *symmetry elements*, which can be axes of rotation, mirror planes of reflection or points of inversion. For example, a uniform sphere can be rotated around its central axis by any number of degrees, yet an observer would not be able to distinguish the sphere before and after the rotation. The number of possible symmetry operations that can be done for a given object is known as the *order* $|\mathcal{G}|$ of the object, and can be seen as a quantisation of 'how symmetric' an object is, where a higher order equates to higher symmetry.

Symmetry element	Notation
Identity	E
Rotation axis	C
Reflection plane	σ
Inversion center	i
Rotation-reflection axis	S

Table 3.1: Symmetry elements are denoted using Schönflies notation.

The primary notation method for symmetry elements is called *Schönflies notation*, and is shown in Table 3.1. First in the list is the identity element E, which applies to every object because it leaves the object completely unchanged. Second, symmetric rotations known as *proper rotations* are done around an axis C, which can be accompanied by a subscript n as C_n to indicate the possible $360°/n$ rotations. In the case that an object has multiple such axes, the one with the largest value of n is called the *principal axis*. Next, planes in which the object can be mirrored symmetrically are denoted by σ. If the plane includes the axis of symmetry of the object (which divides the object in two equal halves) it is called a vertical mirror plane and labelled σ_v. Meanwhile, a plane perpendicular to the axis of symmetry is known as a horizontal mirror plane σ_h. In the specific case that a vertical mirror plane passes between two C_2 axes it is called a dihedral (or diagonal) mirror plane and is labelled σ_d. Moreover, the inversion center i defines a point in the object through which all points of the object can be reflected into one another. It is therefore a center of symmetry that leaves the object invariant under inversion. Lastly, rotation-reflection operations consist of a rotation by $360°/n$, followed by a reflection in a plane perpendicular to the rotation axis. This operation is also called *improper rotation*, and is denoted by S_n.

To understand the distinction between the abstract and point groups, it is useful to consider the *group table* of a simple group. A group table is nothing more than a unique representation of all the possible symmetry operations of an object. Example tables for an object with abstract group $Z/3$ with corresponding point group C_3 is

$\oplus$	0	1	2
0	0	1	2
1	1	2	0
2	2	0	1

$\times$	1	r	r^2
1	1	r	r^2
r	r	r^2	1
r^2	r^2	1	r

Table 3.2: The difference between the abstract group $Z/3$ (left) and point group C_3 (right) is a matter of notation. The point group provides geometric meaning to the transformations as rotations in space.

shown in Table 3.2. As can be seen in the table the difference between the abstract and point groups of an object are a matter of notation and geometric significance. For example, the abstract cyclic group $Z/3$ additive table is the equivalent of point group C_3 multiplicative table, but the point group has more information. Namely, C_3 realises the group table as isometric transformations of space, specifically rotations denoted R, or R^2 for two sequential rotations of $360°/3 = 120$ degrees. That is why you can have several different point groups that realise the same abstract group. For example, D_6, C_{6v}, D_{3d} and D_{3h} are all exactly the same abstract group, but represent different point groups. This is due to the fact that point groups also carry information on how the object isometries transform space, while the abstract group is the group of operations that leave the system invariant.

Conveniently, it is possible to visualise all symmetry elements using the point group of an isolated fullerene hexagonal ring, which contains each symmetry element as shown in Figure 3.12. Naturally, the complexity of point groups quickly increases as molecules get larger and larger. In fact, in three dimensions there are an infinite number of point groups, but for all polyhedral objects they can all be classified by a handful of families which are shown in Table 3.3. The largest possible group is the icosahedral group I_h, which is the maximal symmetry a polyhedron can achieve. For fullerenes, the C_{60} Buckeyball is an example of a molecule with the I_h point group, making it maximally symmetric. Therefore, the point group of a fullerene isomer will always be either I_h or a subgroup thereof. In total, it can be shown that there are 28 possible point groups for fullerenes (Deza et al., 2009), which are sorted according to their order in Table 3.4.

It should be noted that there are infinitely many fullerenes with I_h symmetry, but only one for each N-atom fullerene isomer space, if it exists in the space. Every such isomer of I_h or I symmetry can be found by repeatedly applying operations known as *Goldberg-Coxeter* transforms to a C_{20}-I_h fullerene (Goldberg, 1937), more on which can be read about in Schwerdtfeger et al. (2015, pp. 105–111).

The relation between the geometry and symmetry of the icosahedron can be best understood by going through all the symmetry elements about which the operations in the point group are done. Starting, it has a center of inversion because of its spherical nature. Moreover, there are 6 C_5 proper rotation axes that pass through the centers of the six pairs of opposite pentagonal faces, leading to $6 \times 4 = 24$ C_5 symmetry operations in the point group. Similarly, there are 10 C_3 axes for the pairs of hexagons, 15 C_2 passing through pairs of the carbon-carbon bonds, 15σ mirror planes that stretch between the $30/2$ edge pairs, $6S_{10}$ improper rotation axes through pairs of co-planar pentagons that are oriented at $36°$ from each other, and lastly 10 S_6 axes for the pairs of hexagons as well.

Name	Notation	Order	Symmetry operations
Dihedral group	D_n	$2n$	E, C_n, nC_2
	D_{nh}	$4n$	$D_n, \sigma_h, n\sigma_v$
	D_{nd}	$4n$	$D_n, n\sigma_d$
Tetrahedral group	T	12	$E, 3C_2, 8C_3$
	T_d	24	$T, 6S_4, 6\sigma_d$
	T_h	24	$T, i, 8S_6, 3\sigma_h$
Octahedral group	O	24	$E, 6C_4, 8C_3, 9C_2$
	O_h	48	$O, i, 8S_6, 6S_4, 3\sigma_h, 6\sigma_d$
Icosahedral group	I	60	$E, 24C_5, 20C_3, 15C_2$
	I_h	120	$I, i, 24S_{10}, 20S_6, 15\sigma$

Table 3.3: The various point groups of polyhedrons with their respective symmetry elements.

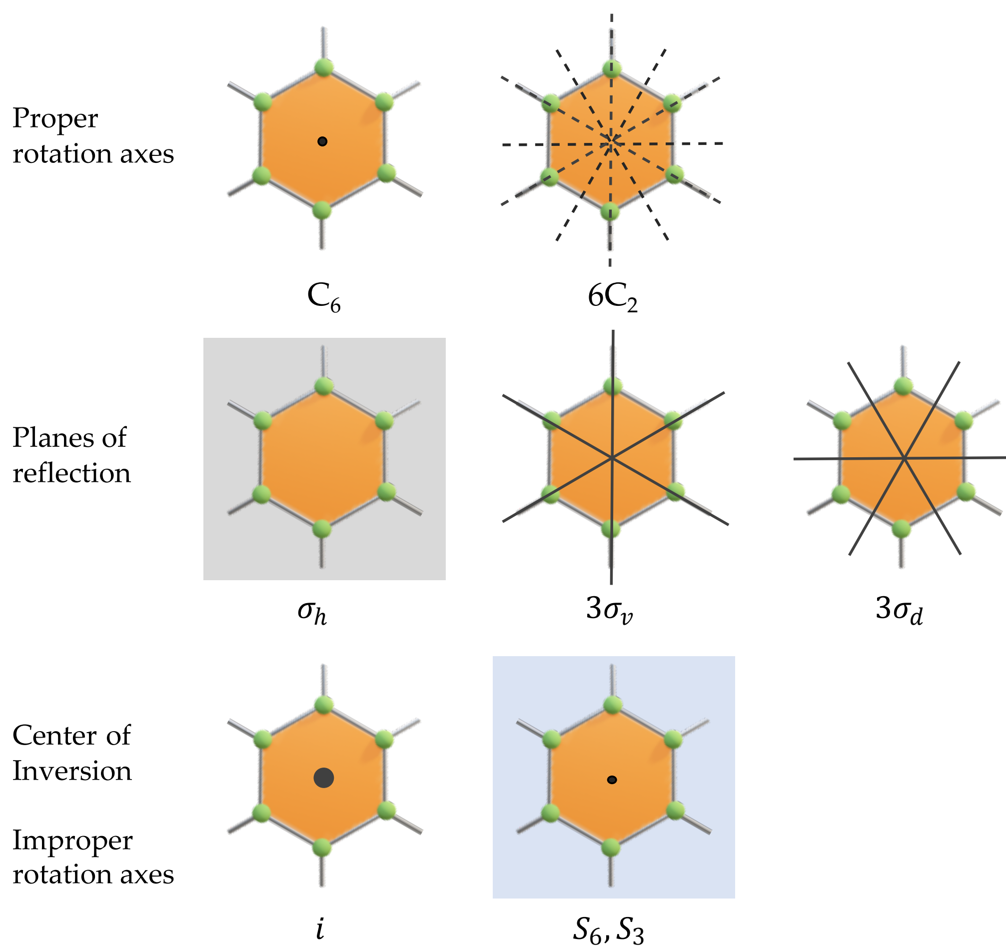


Figure 3.12: The symmetry group of the fullerene hexagon ring contains all types of symmetry elements.

Order	Point Groups	Order	Point Groups	Order	Point Groups
120	I_h	12	T, D_6, D_{3h}, D_{3d}	4	D_2, S_4, C_{2h}, C_{2v}
60	I	10	D_5	3	C_3
24	T_d, T_h, D_{6h}, D_{6d}	8	D_{2h}, D_{2d}	2	C_2, C_S, C_i
20	D_{5h}, D_{5d}	6	D_3, S_6, C_{3h}, C_{3v}	1	C_1

Table 3.4: The 28 possible point groups for fullerenes sorted according to their order $|\mathcal{G}|$ of the group $\mathcal{G}$. From Schwerdtfeger et al. (2015, p. 109).

3.6.2 *Precursor-unfolding symmetry*

Perhaps unsurprisingly, the point group of a planar precursor-unfolding is a planar subgroup of the point group of the target fullerene. Therefore, the highest possible symmetry a precursor-unfolding can have is the a planar subgroup of I_h, which are the D_6 or C_6 point groups of order 6 and 12. The difference between the two types is depicted using simple planar objects in Figure 3.13. As can be seen, the planar C_{60}-I_h precursor molecule from earlier has C_3 symmetry, while the point group of the planar hexagonal fullerene ring of Figure 3.12 is D_6. There are no reflections through the horizontal plane possible in planar symmetry, but a dihedral group does introduce $n\sigma_v$ mirror planes.

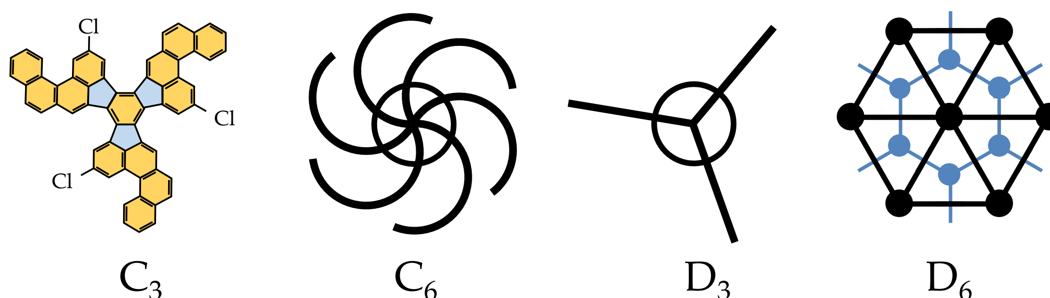


Figure 3.13: A dihedral group D_n has an n reflection symmetries in addition to the n cyclic rotation symmetries the C_n group has.

If the target fullerene exhibits one of the other 28 possible subgroups of I_h, the planar subgroup for the precursor does but not necessarily need to be a subgroup of D_6 as well, because D_n and C_n with $n = 2, 3$ or 5 are also valid possibilities. Moreover, D_6 symmetry is not limited to fullerene isomers that have the I_h point group, as the hexagonal ring with point group D_{6h} and planar subgroup D_6 shows. In other words, a maximally symmetric unfolding does not necessarily fold up to a maximally symmetric fullerene.

3.7 COMPUTING SYMMETRY

To generate maximally symmetric precursor-unfoldings, the point group of the target fullerene isomer needs to determined first, as the planar subgroup of the unfolding can be derived from it. For general graphs, automatically determining the point group without referring to spatial coordinates, i.e. without having to create a 3D polyhedral embedding of the bond-graph first, is difficult to do. However, a remarkable theory by Mani (1971) states that any polyhedral graph can be embedded as a convex polyhedron in 3D space such that the abstract group of the graph (also called the *graph automorphisms*, i.e., renaming of vertices) are realised on the 3D polyhedron as point group operations, which, in 3D are the isometries: rotations, reflections, and inversions. The fact that all the abstract operations for the cubic polyhedral bond-graphs of fullerenes are conceivable as point group operations is non-trivial, but allows for efficient and relatively simple calculation of the point group of any target isomer.

As it turns out, the generalised spiral algorithm from Wirz et al. (2018) is also able to compute the abstract group $\mathcal{G}$ of a fullerene isomer given its dual bond-graph G^*. It does so by making clever use of the fact that all the graph automorphisms can be found by seeing which spiral starts, consisting of three faces (f_1, f_2, f_3), unwind G^* to the complete spiral S. Each of the possible spiral starts that do so is a symmetry operation from the abstract group, making the number of possible starts the order $|\mathcal{G}|$ of the abstract group. From this, the algorithm is able to build the group table by composing all pairs of operations together.

Once the algorithm has created a representation for the abstract group, it is able to use the collection of graph automorphisms from the group to identify the point group of the given fullerene molecule. It does so by looking at all the possible permutations (i.e. vertex shuffling operations) of the so called *symmetry points*. For a fullerene, the symmetry points of interest are the vertices, midpoints of edges, as well as the barycenters of the polygonal faces. The algorithm then considers all the possible symmetry operations it can do at each of the symmetry points, which can

be grouped in so called *site-symmetry* groups. Then, by identifying and counting the number of sites k that belong to each of the site-symmetry groups, it is possible to get a signature that is unique for each possible point group. The signature consists of a sum of permutation counts for each type of site-symmetry:

$$m_s(k) = m_F(k) + m_E(k) + m_V(k) \tag{3.10}$$

where m_F stands for the count of the face site-symmetries, similarly m_E for the edge midpoints, and m_V for the vertices. As a last step, the total count $m_s(k)$ can be fed to the decision tree structure found by Fowler (2006), shown in Figure 3.14. Using the tree, the point group for any fullerene isomer can be determined using the site-symmetry count. Further details on the generalised spiral algorithm that is able to determine the point groups can be found in Wirz et al. (2018).

Figure 3.14: The decision tree for determining the point group of any fullerene from the group order and orbit counts. From Schwerdtfeger et al. (2015, p. 111).

4 THE ALGORITHM

This chapter explains how the theory established in the previous chapters is used to develop algorithms able to produce planar precursor molecules. After the research problem is defined, the following section elaborates on the design of the algorithm. Here it becomes clear what computational steps are required to generate a planar precursor molecule, and what considerations are needed in each of them. The last section deals with the Python implementation of the designed algorithm, and will discuss the code to explain the inner workings of the algorithm in detail.

4.1 PROBLEM STATEMENT

The core focus of this project is to develop algorithms that can successfully and reliably generate planar precursor-molecules that lie at the basis of the autoassembly process. Therefore, the main goal of the project is to write new software that:

P.1 Generates unfoldings

- (a) Generates a carbon fullerene precursor molecule unfolding from nothing but the fullerene bond-graph.
- (b) Generates all possible unfoldings for any N-atom fullerene isomer given enough computation time.

P.2 Encodes generation constraints

- (a) Generates unfoldings that are comprised of full cubic faces, meaning hexagons and pentagons.
- (b) Generates unfoldings with (maximal) planar symmetry.

4.2 ALGORITHM DESIGN

This section discusses the required algorithm components that build a solution to the problem statements and elaborates on design decisions along the way. First, the mechanism behind the generation of a single unfolding is discussed, considering what the available data (input) and the desired results (output) are. In this case, the input is exclusively the fullerene bond-graph, and the output a precursor-molecule unfolding on the Eisenstein plane. Moreover, it is required to establish a method of getting from the input to the desired output that takes into account any necessary constraints needed to make a valid precursor-molecule. Afterwards, the algorithm can be expanded to generate all unfoldings as per P.1b, and encode generation constraints to solve for P.2. A simplified version of the general idea behind the algorithm is shown in Figure 4.1.

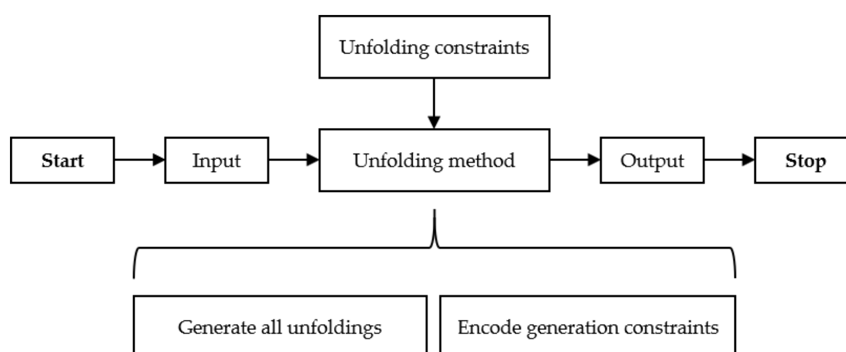


Figure 4.1: Diagram of the general algorithm design, with steps to design the input, output, and method to get from one to the other. Subsequently, functionality to generate all unfoldings and encode generation constraints can be developed.

4.2.1 Designing P.1a: Input

The objective of the project is to exclusively use the fullerene bond-graph to produce precursor-molecule in the form of unfoldings. The main requirement for valid molecules is to have all the atoms placed exactly once on the Eisenstein plane. Because of this, it was chosen to represent the input data as a sparse adjacency matrix, since it both describes all the atoms from all the faces of the molecule and defines the clockwise oriented surface. A simple way to represent all the faces from

the precursor-unfolding in the same matrix is to store the twelve pentagons at the top rows with a dummy value in its sixth column, while the remaining rows contain the hexagons with all six columns filled. Define the *dual-neighbours* array as the described data structure that stores the bond-graph in a $N_f \times 6$ array with N_f the number of faces in the precursor-molecule. The array is illustrated in Figure 4.2, with the pentagons stored in the top rows and the $N_f - 12$ hexagons thereafter. Compared to the full adjacency matrix, the sparse matrix saves in both required storage and computation time. Storage space is saved as there are fewer elements in the array, and thus fewer to store in memory. There is moreover a reduction is computation time as any function that has to looks up elements in the array has a smaller number of elements to sift through.

Having figured out the desired representation, the choice remains to either use cubic- or dual-graph data as the input for the algorithm. The fullerene dual was chosen as it offers several advantages over the cubic graph. First of all, because the dual graph and the Eisenstein plane allow for the use of exact integers which greatly simplifies the algorithm. This is in contrast to the less intuitive numerics that would have to be used for the cubic graph.

$$\underset{(N_f \times 6)}{\text{dual-neighbours}} = \begin{bmatrix} n_{0,0} & n_{0,1} & n_{0,2} & n_{0,3} & n_{0,4} & \uparrow \\ n_{1,0} & n_{1,1} & \cdot & \cdot & \cdot & 12 \\ \vdots & & \ddots & & & \downarrow \\ n_{12,0} & \cdot & \cdot & \cdot & \cdot & n_{12,5} \\ \vdots & & & & \ddots & \vdots \\ n_{N_f,0} & \cdot & \cdot & \cdot & \cdot & n_{N_f,5} \end{bmatrix}$$

Figure 4.2: The input is designed as a $N_f \times 6$ sparse adjacency matrix with each row containing the dual neighbours of the index node 0 to N_f. The twelve pentagons are stored in the top rows with a dummy value in the last column.

4.2.2 *Designing P.1a: Output*

Given the dual-neighbours array as input, the desired output becomes a dual-graph unfolding of the precursor-molecule onto the Eisenstein plane, whose equilateral triangles indirectly describe the locations of the atoms at each triangle centroid. This creates two options for storing the unfolding: either the coordinates of the atoms are directly stored in an array, or they are stored indirectly by instead putting the triangle coordinates in the array. The first option, albeit clear and direct, does not show the orientation of the surface, obscuring information about which atoms will form bonds during autoassembly. Moreover, although the locations of the atoms can be obtained from the triangles, the triangle information cannot be retrieved from the coordinates of the atoms. In other words, representing an unfolding in terms of atoms instead of the dual graph nodes and edges causes a clear loss in information.

Knowing that the unfolding is best stored as the dual-graph triangles, the question becomes how to represent them in a storage array. A first intuition might be to simply store all the node coordinates from all the triangles. However, this would again obscure knowledge on which arcs align during autoassembly. Moreover, knowing the orientation of the precursor-molecule surface makes everything much easier. Therefore, arcs need to be stored as they both give the location of the nodes as well as the direction between them.

At this point the question becomes whether to store full triangles or single arcs in the storage array. A triangle element would be of the form:

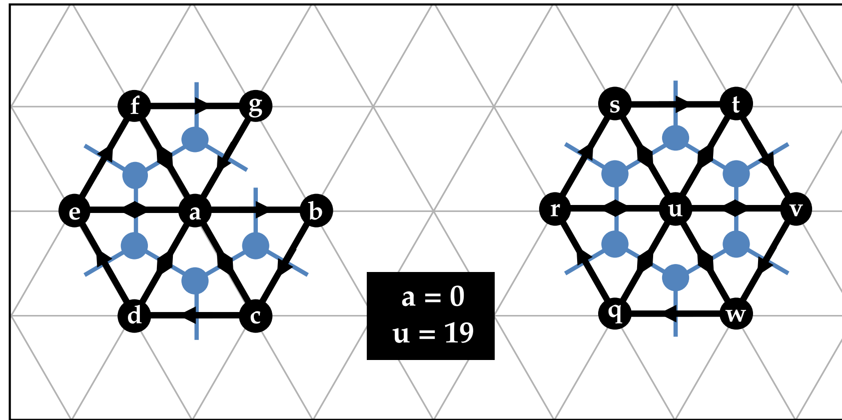
$$[[u, v, w], [i_u, j_u], [i_v, j_v], [i_w, j_w]]\ ,$$

while the same triangle would take three elements in terms of single arcs:

$$\begin{bmatrix} [u, v], [i_u, j_u], [i_v, j_v] \\ [v, w], [i_v, j_v], [i_w, j_w] \\ [w, u], [i_w, j_w], [i_u, j_u] \end{bmatrix}$$

Although storing triangles saves space, it does make it more difficult to find the arcs that align during autoassembly, i.e. $[u, v]$ and $[v, u]$. After all, with a single arc $[u, v]$ its corresponding reverse could be found by going through the array once and looking for $[v, u]$, whilst per triangle there are now three sub-arrays to look for: $[u, v, w]$, $[v, w, u]$ and $[w, u, v]$ from which the arcs need to be extracted. Finding the arcs that glue together is an important feature for the ease of autoassembly, and was therefore prioritised over the reduction in storage space. Additionally, the triangles can always be formed from the collection of arcs, since each arc is part of a unique triangle.

One way to store all the arcs would be to make an array of $N_{arcs} \times 1$ where every arc has a unique location in the array. Another would be to instead create a $N_{triangles} \times 3$ array where each row contains the three arcs that make up a triangle. However, a more sophisticated structure can be made that used the shape of the input as a $N_f \times 6$ array in which each arc is an element with a unique location in the array. Defining this structure as the *arc-array*, it has the added benefit that the nodes don't have to be stored explicitly anymore. To understand this, consider how each row of the arc-array corresponds to a face of the precursor-molecule. Moreover, consider that each entry in the row represents a unique arc, which in turn defines a triangle that is part of the face. As such, the dual node 0 represents the cubic face 0, which means its dual graph triangles can be stored in the top row of the array. Similarly, every triangle that is part of face 19 with dual node 19 at its centre can be stored in the 19th row. The final data structure with example storage of a pentagon and hexagon is shown in Figure 4.3. Moreover, the other considered structures are shown in Figure 4.4 for visual reference.



$$
\underset{(N_f \times 6)}{\text{arc-array}} =
\begin{array}{r}
\text{face a = 0} \\ \\ \\ \\ \text{face u = 19} \\ \\ \text{face } N_f
\end{array}
\left[
\begin{array}{cccccc}
[a,b] & [a,c] & [a,d] & [a,f] & [a,g] & \uparrow \\
\vdots & \vdots & \vdots & \vdots & \vdots & 12 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \downarrow \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
[u,v] & [u,w] & [u,q] & [u,r] & [u,s] & [u,t] \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
\right]
$$

Figure 4.3: The arc storage array is designed as a $N_f \times 6$ array with arcs $[a, b] = [i_a, j_a], [i_b, j_b]$ as array elements and the arc nodes implicit in the array structure. The example pentagon a and hexagon u are shown together with their storage in the arc-array for the case where $a = 0$ and $u = 19$.

$$
N_f
\begin{bmatrix}
\text{arc}_{0,1} & \cdots & \text{arc}_{0,5} & \uparrow \\
\vdots & \vdots & \vdots & 12 \\
\vdots & \vdots & \vdots & \downarrow \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\text{arc}_{N_f,1} & \cdots & \cdots & \text{arc}_{N_f,6}
\end{bmatrix}
\quad
\begin{bmatrix}
\text{tri}_{0,1} & \cdots & \text{tri}_{0,5} & \uparrow \\
\vdots & \vdots & \vdots & 12 \\
\vdots & \vdots & \vdots & \downarrow \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\text{tri}_{N_f,1} & \cdots & \cdots & \text{tri}_{N_f,6}
\end{bmatrix}
\quad
\begin{bmatrix}
\text{atom}_{0,1} & \cdots & \text{atom}_{0,5} & \uparrow \\
\vdots & \vdots & \vdots & 12 \\
\vdots & \vdots & \vdots & \downarrow \\
\vdots & \vdots & \vdots & \vdots \\
\vdots & \vdots & \vdots & \vdots \\
\text{atom}_{N_f,1} & \cdots & \cdots & \text{atom}_{N_f,6}
\end{bmatrix}
$$

(a) ✓ **(b)** ✗ **(c)** ✗

Figure 4.4: The considered designs for the output by storing either arcs **(a)**, triangles **(b)** or atoms **(c)**. **(a)** was chosen because of its ease of use and possible transformation into **(b)**. **(c)** was rejected because of the loss in surface orientation information.

4.2.3 Designing P.1a: Unfolding updates

With the input and output defined, the question becomes how to get from the dual bond-graph to a filled array of unfolding arc coordinates. First, it is important to note that arcs in and of themselves have no physical significance, as only complete triangles represent an atom. Because of this, the method for creating an unfolding should be to place triangles one after another on the Eisenstein plane until all triangles, and therefore all arcs in the output array, are accounted for.

There are several things to consider in going about initialising the placement process and in handling subsequent placement steps. The first is that triangles cannot simply be placed in random locations on the grid after one another. This is due to the fact that, as was discussed before, triangles are not allowed to be disconnected from one another on the grid as it would lead to unpredictable reactions during autoassembly. Therefore, new triangles should always be appended to the part of the precursor-unfolding that is already placed. This has two implications: the first being that a *root-triangle* needs to be defined, whereupon further triangles can be appended. Although the choice of root-triangle does influence the isomer you generate, it does not affect the generality of the unfolding algorithm when generating all possible unfoldings, because all possibilities of triangle configurations will be created. Therefore, the triangle that is placed first is inconsequential so long as some triangle is placed to start with.

The second implication is that new triangles can only be formed from arcs that share nodes with the triangles that are already placed. In other words, new triangles need to be formed from *reverse arcs* of the existing triangles so as to attach them to the periphery of the existing incomplete unfolding. This is exemplified in Figure 4.5 for a handful of triangles, where it is shown how only a specific set of triangles (represented by dotted arcs) is eligible for placement at this stage of the unfolding process. The algorithm will also need to be able to determine the unknown nodes and arcs of a new triangle when given a peripheral arc.

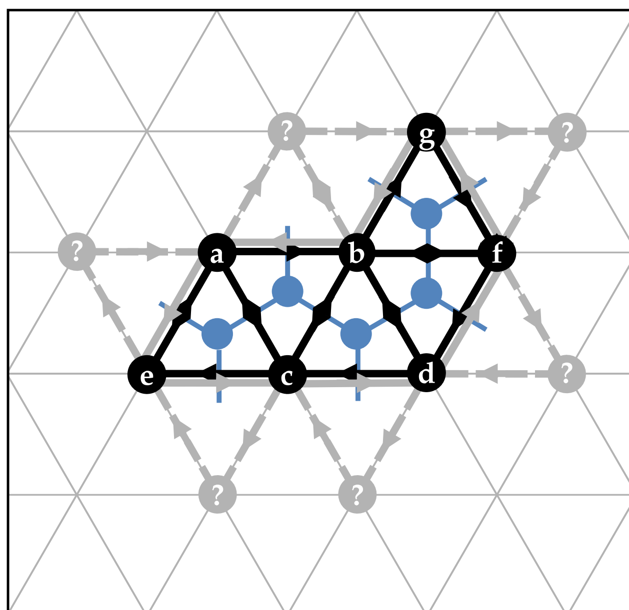


Figure 4.5: The only possible triangles that can be placed at any point in the unfolding process are those defined by the arcs on the periphery (blue) of the incomplete unfolding (black). The unknown nodes marked by question marks can be determined from the dual bond-graph.

An important design aspect that follows from the second implication is that there is a need for a data structure that contains the set of reverse arcs that lie on the periphery of the existing molecule able to initiate the placement of new triangles. To this end, a *workset* can be defined that is a queue of arcs, where the algorithm can take a *work-arc* from the top of the queue at each placement step to grow the precursor-unfolding, and append newly found work-arcs on the periphery of the triangle to the bottom of the queue. In this way, starting from the root-triangle, arcs are assigned Eisenstein coordinates, reversed, added to the set and so on. A diagram of the placement cycle of a triangle is shown in Figure 4.6.

An accompanying visualisation of triangle placement is shown in Figure 4.7. The placement starts with an arbitrary work-arc $[u, v]$ taken from the workset with Eisenstein coordinates $[i_u, j_u], [i_v, j_v]$. Then, the third triangle node w has to be found together with its coordinates. Because of the way the dual-neighbours array is structured, the node is found in row u as the next node at depth v. After all, a particular row in the array is a clockwise cycle of neighbours to a centre node, such

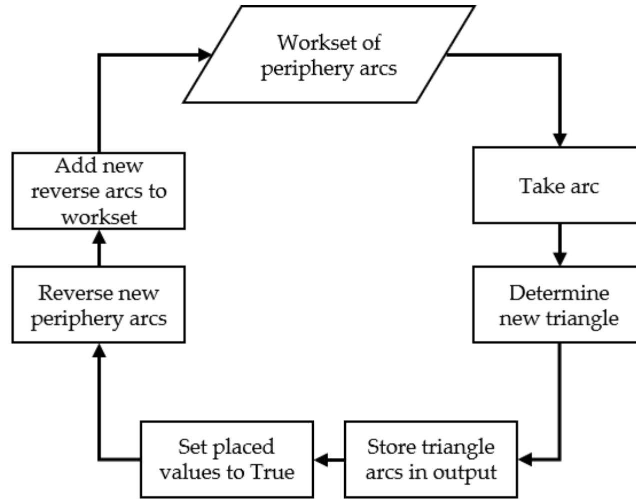


Figure 4.6: Diagram of the design of the unfolding method, where triangles are placed, reversed, and a set of arcs is maintained to continue the process.

as u in the figure. In this way, the node next to v in the row is the next clockwise neighbour of u on the face. After the node is determined, the coordinates can be found as depicted in Figure 4.7. First, the current direction of the arc is determined by translating it onto the Eisenstein ring, which can be done by subtracting the coordinates from end node v from those of the starting node u. Second, the current direction should be Eisenstein multiplied according to Equation (3.9) in clockwise direction to find the new direction. This marks the coordinates of w on the Eisenstein ring relative to the starting node u. Lastly, the Eisenstein ring coordinates are summed with the original coordinates of the triangle nodes, which translates them back to the intended placement location in the precursor-unfolding. Mathematically, the three steps result in the following equation:

$$[w_i, w_j] = [u_i, u_j] + ([v_i, v_j] - [u_i, u_j]) \cdot [1, -1], \tag{4.1}$$

where the dot represents Eisenstein multiplication with $[1, -1]$ the clockwise direction.

To visualise how the workset is updated and an unfolding can be created, consider Figure 4.8. In the figure, first a root-triangle is placed and its reverse arcs are added to the workset, where each element of the queue is an arc together with its coordinates.

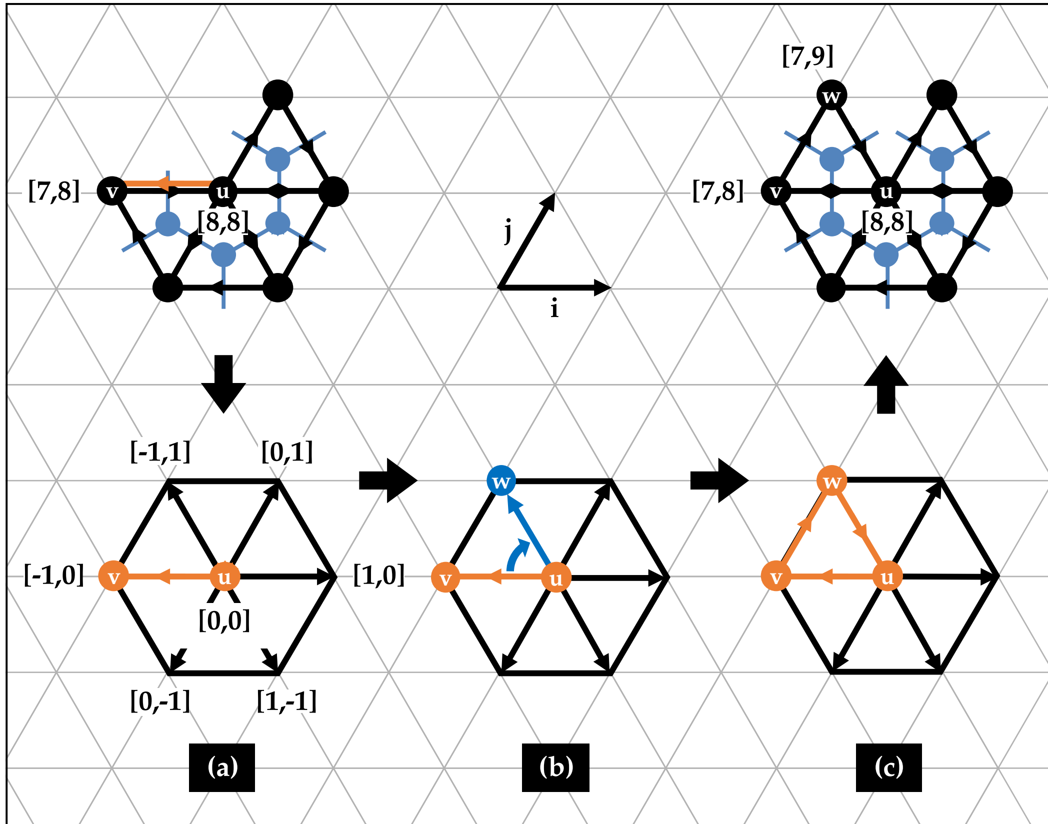
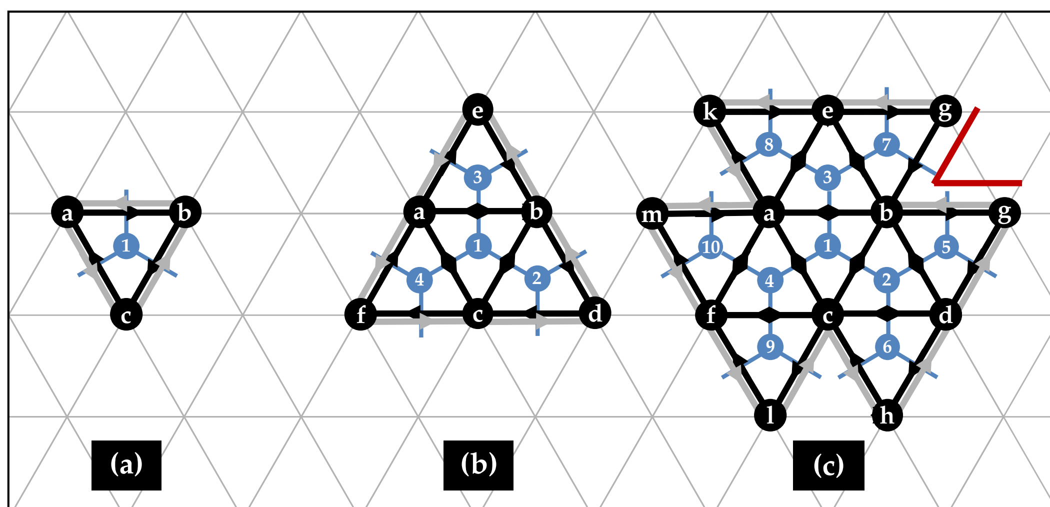


Figure 4.7: **(a)** A work-arc is taken from the workset queue; **(b)** uses Eisenstein multiplication to find the third node of the triangle and form the arcs; **(c)** returns the full triangle for placement on the Eisenstein plane.

After three cycles using those initial work-arcs, three new triangles are created which each add two work-arcs to the workset. At this point the root-triangle work-arcs are no longer in the set. Lastly, after six more placement steps it can be seen how single triangles start forming faces, for example the pentagon centred by node b. As can be confirmed by the reader the hexagons centred by nodes a and c can be formed in future placement steps.



$$\begin{bmatrix} [c,b] & [i_c,j_c] & [i_b,j_b] \\ [b,a] & [i_b,j_b] & [i_a,j_a] \\ [a,c] & [i_a,j_a] & [i_c,j_c] \end{bmatrix} \quad \begin{bmatrix} [d,b] & [i_d,j_d] & [i_b,j_b] \\ [c,d] & [i_c,j_c] & [i_d,j_d] \\ [e,a] & [i_e,j_e] & [i_a,j_a] \\ [b,e] & [i_b,j_b] & [i_e,j_e] \\ [f,c] & [i_f,j_f] & [i_c,j_c] \\ [a,f] & [i_a,j_a] & [i_f,j_f] \end{bmatrix} \quad \begin{bmatrix} [d,g] & [i_d,j_d] & [i_g,j_g] \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots \\ [a,m] & [i_a,j_a] & [i_m,j_m] \end{bmatrix}$$

Figure 4.8: (a) The workset is updated with the reverse arcs (grey) of the root-triangle $[a, b, c]$. **(b)** The resulting triangles are again reversed and peripheral arcs added to the workset. **(c)** A pentagon is formed at the cutout wedge (red) and more triangles are placed. The triangle atoms are numbered in order of placement.

4.2.4 *Designing P.1a: Unfolding constraints*

It is important to evaluate the unfolding process from Figure 4.6 and see at which points it might produce placement steps that are in conflict with the production of a valid unfolding. Again, the key requirement for the process to produce valid molecules is that each unique atom and therefore triangle can only be placed once. For this reason, another design requirement is to keep track of which atoms, and therefore arcs, are placed at any point in the process. This can be done conveniently thanks to the shape of the output array, as a boolean array can be created that has the same shape as the output array by which each arc in the output corresponds directly

to its boolean value in the placed array. Thus, there is a need to define a *placed-array* of shape $N_f \times 6$ that keeps track of the placement status of each arc in the unfolding. The array should be initialised with all its values set to False, implying no arcs are placed on the grid yet. During the unfolding process, the values in the array can be switched to True when the corresponding arc is assigned Eisenstein coordinates and placed on the grid.

A second constraint is that triangles are not allowed to overlap in the unfolding. It might not be immediately obvious when this could occur, since triangles are only formed at the periphery of the incomplete unfolding. An example of such a collision between triangles is depicted in Figure 4.9, where $[l, n]$ is taken from the top of the workset and the triangle $[l, n, o]$ is considered for placement. However, node o cannot be placed since node g is already occupying the coordinates on the plane. Of course, in the case that nodes l and n actually were the neighbours of g as listed in the dual-neighbours array, there would be no problem in placing triangle $[l, n, g]$.

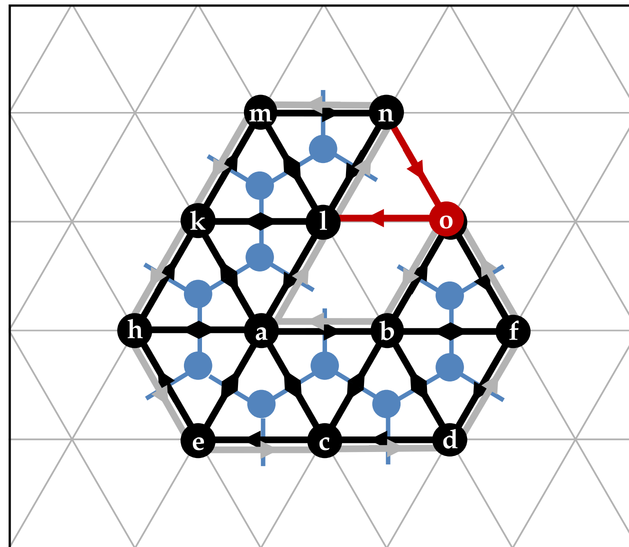


Figure 4.9: An example of how placement steps can result in collision on the Eisenstein grid, necessitating a way to keep track of occupied grid points in the algorithm.

Thus, the algorithm needs to keep track of the occupied grid points on the Eisenstein plane, so that any intended placement coordinates from new arcs can be checked against them. If the grid point is occupied by a different node than the intended

one, the arc (and therefore the whole triangle) has to be rejected. Of course, the arc-array already keeps track of the placed nodes and their coordinates on the grid, but cross-referencing new coordinates with it would take a long time. After all, it would require checking each $N_f \times 6$ elements in the arc-array consisting of two coordinates $[i, j]$ for the source and target of each arc. Instead, defining an associative array called *collision-grid* is more useful. Also known as a dictionary, this data structure is a collection of [key:value] pairs where each key is only allowed to appear once in the collection. In the case of the precursor-unfolding, the key is a specific Eisenstein coordinate $[i, j]$ which may only be occupied once, while the corresponding value is the node placed at that location.

Using the collision-grid, new triangles can be checked node for node to see if their intended placement locations are occupied by other nodes. Moreover, new nodes n from newly placed triangles can be added to the collision grid as

$$[\text{key} : \text{value}] = [[i_n, j_n] : n]$$

It should be remarked that new triangles only add one node to the collision-grid when placed, for example the nodes marked by a question mark in Figure 4.5.

Apart from the convenience that the dictionary data-type cannot have duplicates, finding values in dictionaries is faster than finding them in a list or array. The reason for this is that dictionaries use a hash lookup that has average on $\mathcal{O}(1)$. This is in contrast to the linear lookup runtime of $\mathcal{O}(N)$ using the other data types, where one has to search element by element until the result is found [2]. Especially for bigger fullerenes with hundreds of atoms, the hash lookup will be faster as it remains near constant with $\mathcal{O}(N)$ in the worst case. Meanwhile, the search in lists and arrays will keep growing linearly with the number of occupied grid points as shown in Figure 4.10.

[2]Information on time complexity is available at the official Python Wiki.

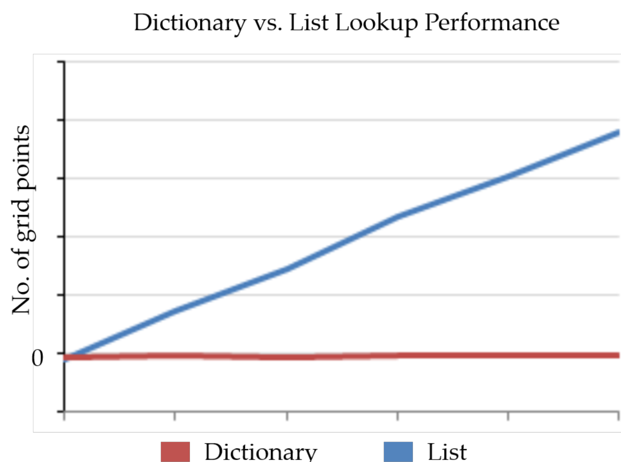


Figure 4.10: The lookup time of a dictionary type is near constant at larger sizes, while the lookup in lists and arrays keeps growing linearly.

It should be noted that the algorithm should never allow the placement of triangles that are already placed, even if there is no collision with the existing molecule as it would result in a re-placement of an atom. Illustrated in Figure 4.11, it is shown how arc $[c, b]$ enters the workset from triangle $1 : [a, b, c]$ as the reverse arc of $[b, c]$. However, due to the order of the workset queue the precursor-molecule grows upwards until $[c, b]$ is placed as part of triangle $7 : [g, c, b]$ with work-arc $[g, c]$. When $[c, b]$ from triangle 1 becomes next in line, the arc would be marked as placed but show no collision on the Eisenstein grid. If the arc would be allowed the resulting placement step would re-place triangle $[c, b, g] = [g, c, b]$ at the other end of the unfolding and overwrite the existing arc-data in the arc-array.

The algorithm is terminated in one of three cases. First, if the unfolding is completed, meaning all triangles are placed on the grid. Second, if there are no more valid placement steps left in the workset, either because it is empty, or because the work-arcs that are left cause a collision or are already placed. Taking the termination conditions and the newly found placement constraints into account, an updated design of the algorithm that can create a single unfolding is shown in Figure 4.12. In it, four phases of creating an unfolding can be identified: to initialise, constrain, update, and terminate the algorithm. For future reference, the *state* of the unfolding can be defined as the current status of the arc-array, placed-array, workset, and

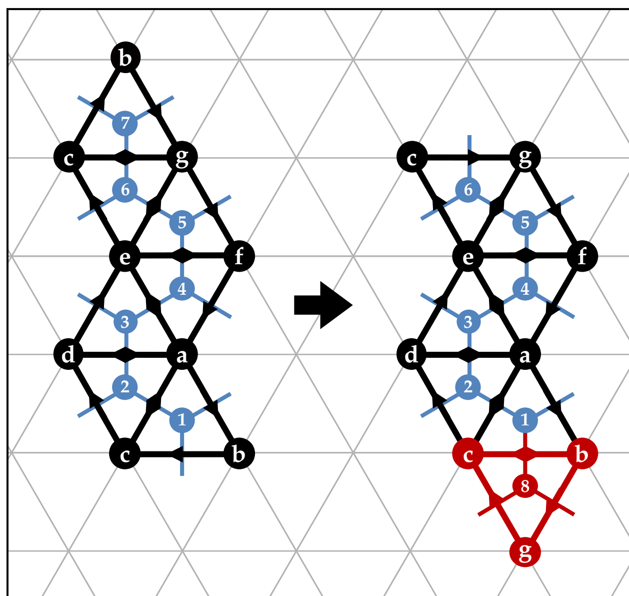


Figure 4.11: An example of an erroneous re-placement of a triangle $[c, b, g]$ which necessitates to check whether a work-arc is exists on the Eisenstein plane before placing its triangle.

collision-grid. The initial state comprises empty output arrays and a workset that contains the root-arc to start the unfolding process, and the final state is a completed precursor-molecule with all arcs stored in the arc-array and the workset empty.

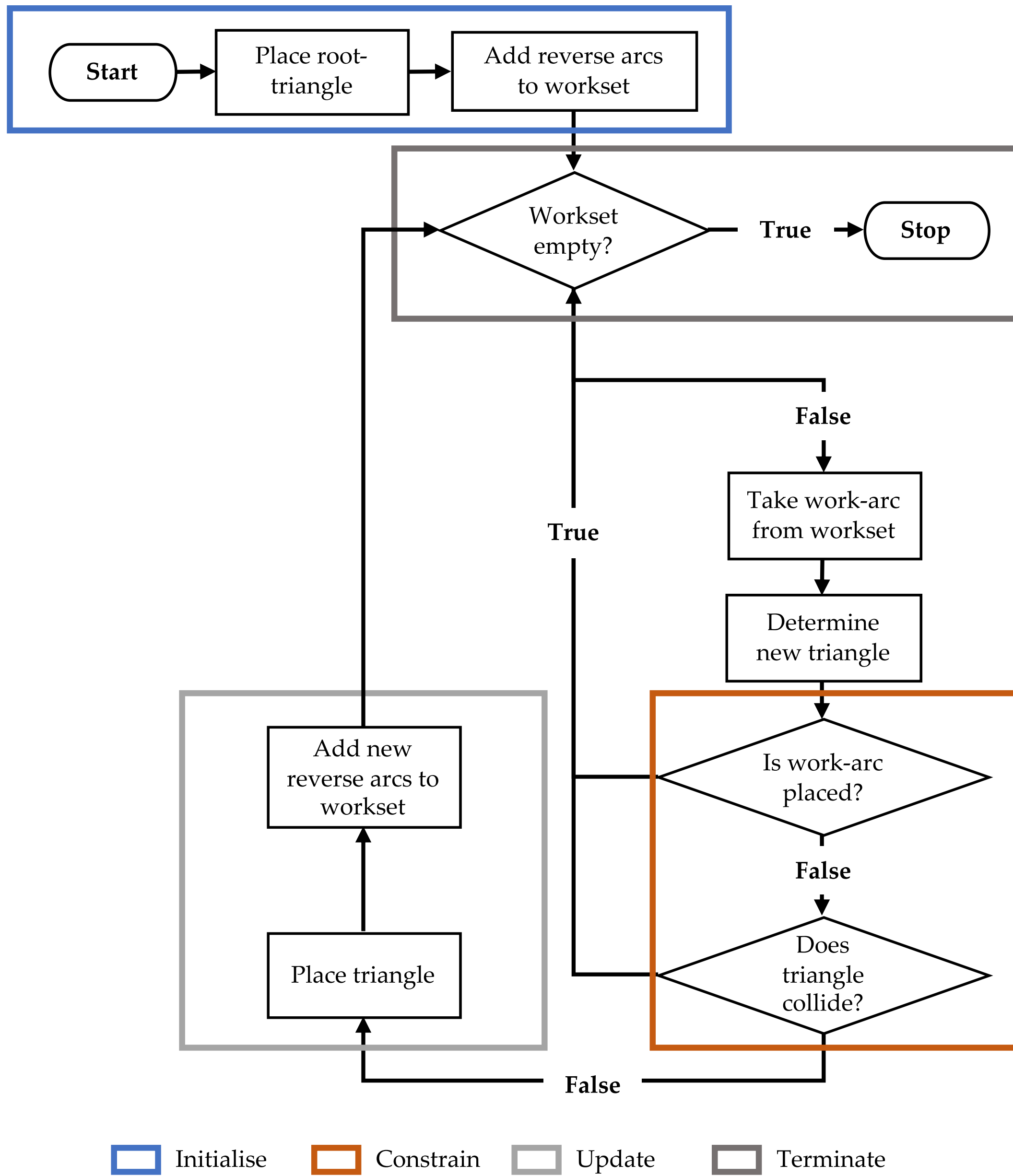


Figure 4.12: Diagram of the processes involved in generating a single unfolding. Four distinct phases in the algorithm can be identified which are to initialise, update, constrain, and terminate the unfolding process.

4.2.5 Designing P.1b: Generating all unfoldings

The next step of the algorithm design is to find a way to generate all possible precursor-unfoldings of any given N-atom fullerene isomer. To this end, note that what influences the configuration of an unfolding is the order in which triangles are placed, and thereby the order in which work-arcs are taken from the workset. Therefore, generating multiple configurations of an unfolding equates going through all possible orderings of work-arcs in the workset queue. However, this has to happen every time the workset is updated with new arcs. In other words, every possible placement option has to be evaluated at every possible placement step. Fortunately, there is a well known type of process known as *recursion* that should be able to exhaustively search through the space of all possible unfolding.

To better understand recursion, consider the metaphor of a squirrel that lives at the root of a *recursion tree*. One day, the squirrel decides it wants to gather all the nuts that grow at the leafs of the tree and bring them back to its home. With an empty bag in hand and starting from the root, the squirrel first climbs as far as possible up the far left branch of the tree. It does so one knot at a time, until it reaches the first leaf at the top of the tree. There, it either finds a nut, finds no nut, or discovers a cut off branch. If the top branch can be reached, the squirrel takes the nut if it is there, puts it in its bag, and climbs back down one knot. Then, its goes on to pick up any existing surrounding nuts from all branches that stem from the knot its standing on. After that, it goes back down another knot, and repeats the process of nut-gathering and backtracking until it has visited all knots and collected all nuts in its bag. In the end the squirrel returns home to the root, carrying with it a bag filled with every nut the tree had on it.

In more formal terms, the squirrel nut-gathering process is called a *depth-first* search on the recursion tree, and is visualised in Figure 4.13. The knots of the tree are known as the *nodes* of the tree, and the depth at which the node lies is called the *recursion level* or *depth*. Moreover, there is distinct nomenclature for the relationships between nodes on the tree. For example, in Figure 4.13, nodes B and C are known

as *siblings*, making A their *parent* as it lies higher in the hierarchy, and D their *uncle* as it is the sibling of parent A. The root-node is the node connected to all child nodes and is also known as the *ancestor*, where all nodes at greater depth are *descendants* of the ancestor. Note that siblings have to be part of the same parent branch, meaning node pairs (A, C) and (E, F) do not share this relationship. Lastly, the nodes at the end of the highest-level branches are called leaf nodes as was the case with the metaphor.

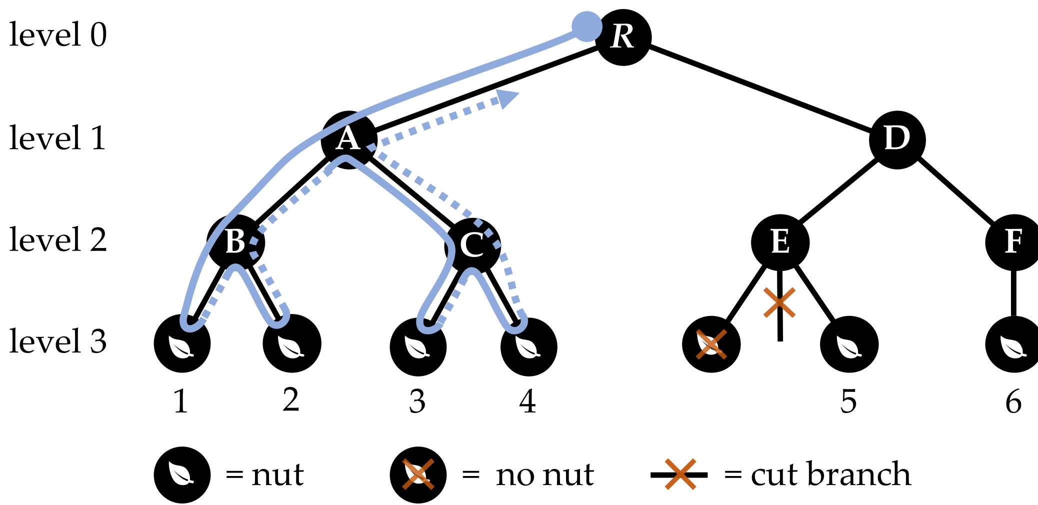


Figure 4.13: From the root R, a depth-first recursion tree squirrel (partial path shown in blue) traverses the tree from node to node until all nodes are visited and all leafs are collected.

Regarding the generation precursor-unfoldings, each node in the tree is analogous to an intermediate state of the unfolding and each nut at a leaf node to a completed unfolding. In this way, each edge between tree nodes represents a placement cycle as per Figure 4.6. Thus, the generation of a single N-atomic unfolding a linear path down the tree from the root-node to an arbitrary leaf node, whereas all possible unfoldings are created by travelling from node to node up and down the tree until all nuts are collected back at the root. Extending the analogy, the root-node corresponds to the placement of the root-triangle, and every child node to the placement of a new triangle that is attached to the existing body of the precursor unfolding. Similarly, a leaf node with a nut marks the placement of the final triangle and therefore the completion of the unfolding.

Exploring all possible unfoldings of the isomer hence implies creating and traversing all possible branches in the recursion tree. The process is visualised for the first two placement steps of an example precursor-molecule in Figure 4.14. What is important to see is how the tree branches represent the creation of unfolding configurations that differ by a single triangle, and that creating all possible branches at each placement step will eventually exhaustively search the entire space of possible unfoldings.

If the vertical traversal of the recursion tree is a result of placing triangles, sibling nodes branching from the same parent are created by trying every periphery work-arc in the workset for placement at each state. To elaborate, consider how the root-triangle adds three work-arcs from its periphery to the workset, creating three branches. In subsequent placement steps, the number of child branches from each parent node is dependent on the length of the workset at that specific state of the unfolding. When placing single triangles one after another, each triangle can at most generate two branches to the tree, since one of the triangle arcs is always attached to an existing periphery arc of the unfolding. This can be seen in Figure 4.14 as well, where each triangle has two periphery arcs that are added to the workset in the order of the numbers next to them. Naturally, the recursive process in the figure can continue onward from the second level, which would create five new branches with child nodes for each of the four states shown at level two.

In terms of computer science, a recursive function is a function that calls itself. It corresponds to mathematical induction, and in the same way can be broken into *base cases* and *recursive cases*. If a base case is reached the function is at a leaf node, where it would subsequently collect the nut if it exists and backtrack to the parent node. In contrast, a *recursive case* occurs for all intermediate states before the base case is reached. To continue the search the function would call itself so that it traverses down a level in the recursion tree and can again evaluate its position in the tree to see if a nut is found. A diagram of the general recursive method is depicted in Figure 4.15. In the figure, a recursion step to deeper levels is called *winding-up*, while a return back to previous calls corresponds to the *unwinding* of the algorithm that occurs once the base case is reached. The unwinding takes place until the function

returns to the very first call and stops. Of course, any statements programmed before heading down a recursion level are executed prior to each call, while any statements set after any recursion step are left pending until the function unwinds back to that particular step. Remark how all wind-up and unwind steps combined represent searching through the whole tree in the same way that the squirrel traverses it in Figure 4.13.

In terms of the unfolding algorithm, a base case is reached (i.e. true) when an unfolding is completed and the nut is ready to be collected. To identify the base case, one can define a variable *tri-count* as the integer number of placed triangles at a given state, so that leaf nodes with a nut can be identified as a state where the tri-count is equal to N. Meanwhile, the base case is false for every recursive case as the unfolding is incomplete with tri-count smaller than N. The base case is thus a basic validity check for generated precursor-molecules to see if all atoms are accounted for.

To create an algorithm that can exhaustively generate all possible unfoldings, the recursion process should thus encapsulate the method from Figure 4.12 by including a base- and recursive case. A renewed diagram of the algorithm that incorporates recursion is shown in Figure 4.16. As can be seen the phases of the single unfolding algorithm: initialise, constrain, update and terminate are the program statements that happen before recursion. Moreover, when the algorithm unwinds back to a previous state, it returns to check if the workset is empty. If there are more placement options (i.e. tree branches) to be explored it starts another path down to a potential leaf node, and does so upon return for all the possible work-arcs in the workset at every node. Should the algorithm terminate at an intermediate state, implying the workset is empty but not all atoms are placed yet, that particular recursion branch will simply not add anything to the list of unfoldings and unwind. This is how branches are ’cut’ from the tree, by simply not having an impact on the current collection of nuts and traversing to the parent node in the tree.

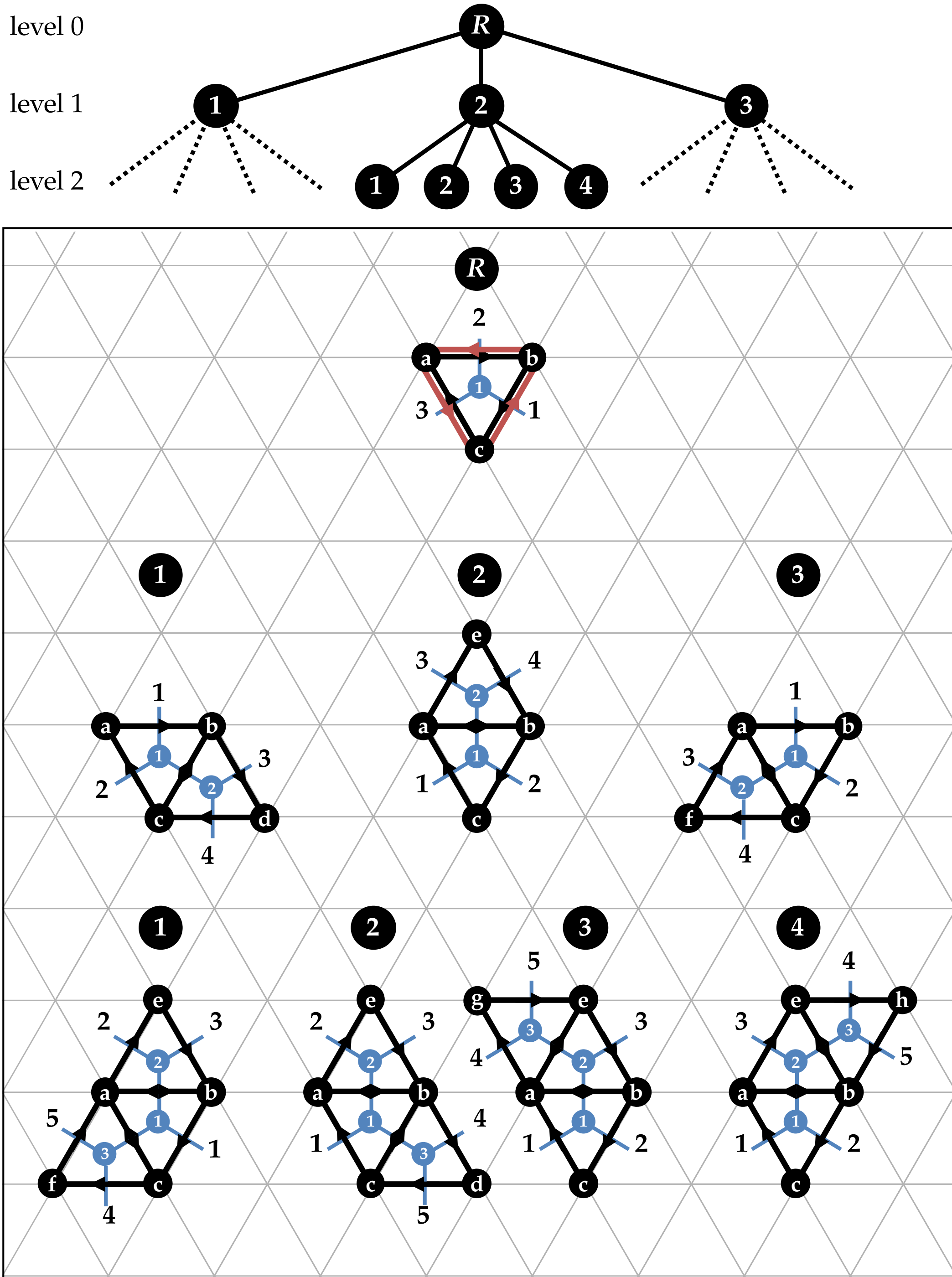


Figure 4.14: Example of how recursion creates different configurations of unfoldings by traversing different branches in the tree.

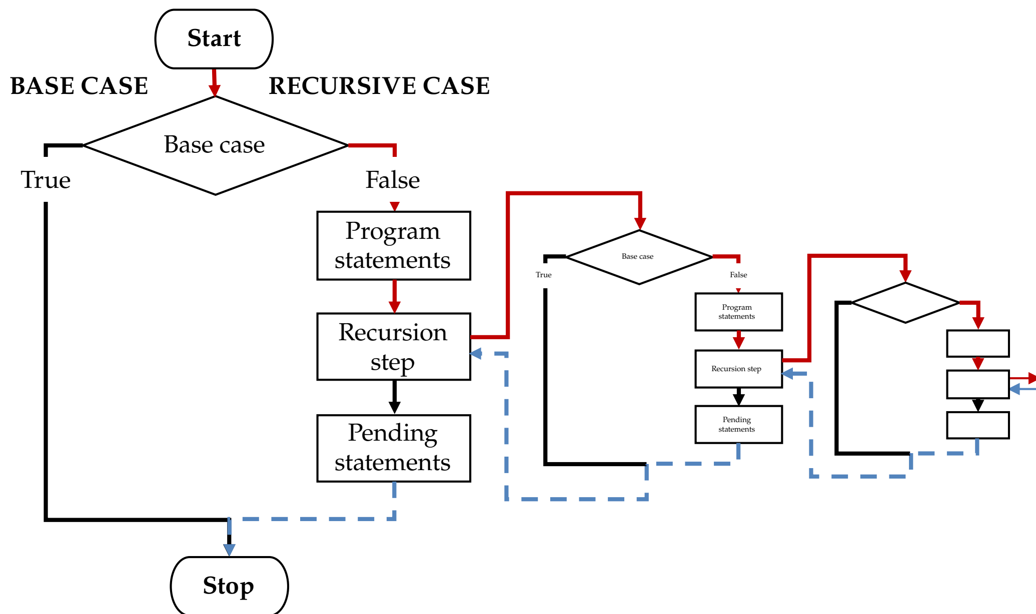


Figure 4.15: Diagram of the general recursion function algorithm, a base case is set, and the function calls itself recursively until it is reached, unwinding in the process.

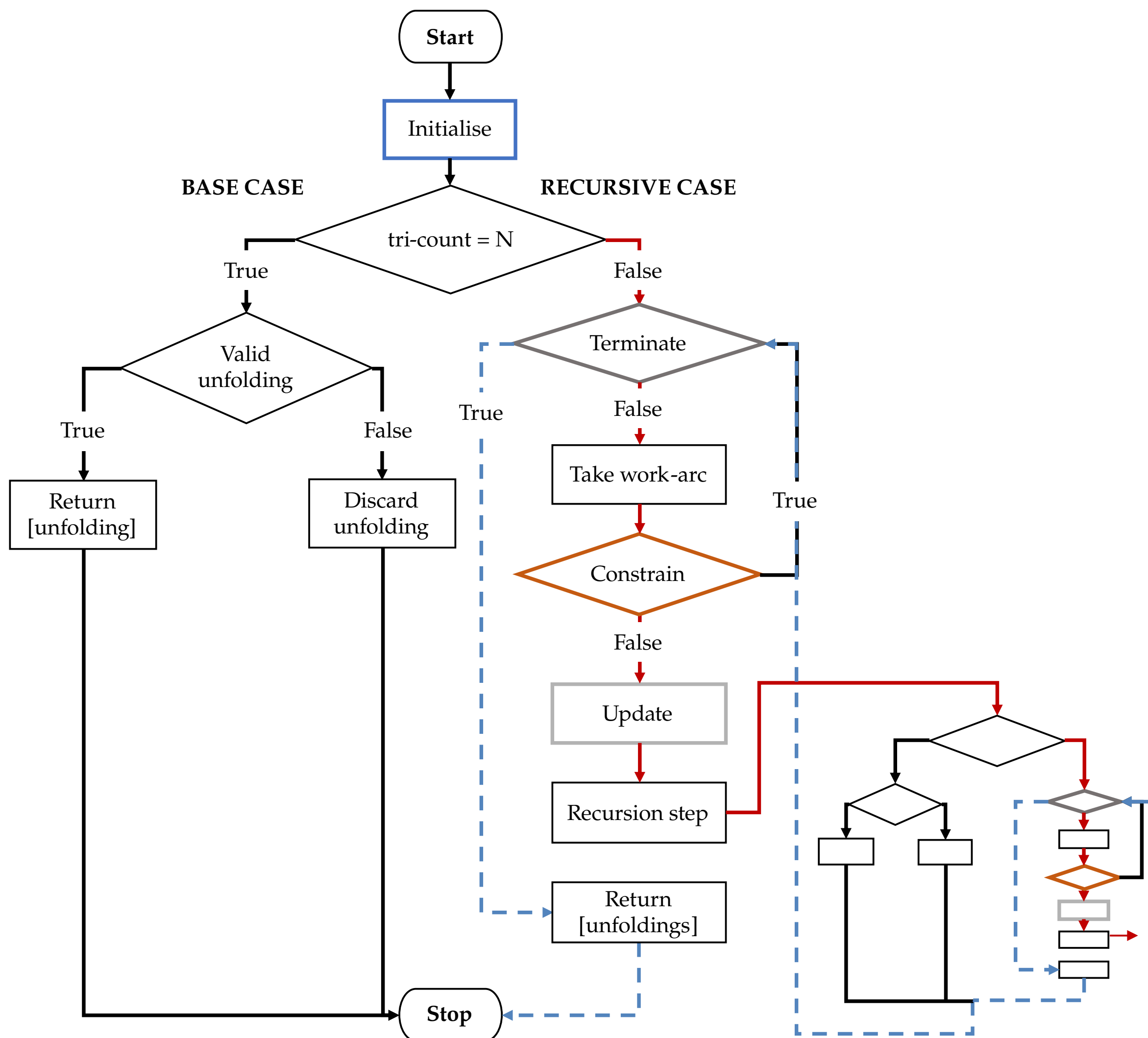


Figure 4.16: Diagram of the complete recursion algorithm that is able to generate all valid possible unfoldings for any given N-atom fullerene by collecting them in a list recursively.

4.2.6 *Designing P.2a: Placing faces*

With a design in place that is able to generate all possible unfoldings, the next step is to create ways to control the shape of the unfolding that could possibly increase the autoassembly success chance of the generated precursor-molecules and reduce the search space. As theorised in the previous chapter, making the molecule consist solely of full hexagonal and pentagonal faces is one way to potentially increase autoassembly stability. Faces, meaning pentagons and hexagons, consist of one unique triangle per atom. Each adjacent triangle shares an edge, but the oppositely directed arcs that make the edge are only part of one of the two adjacent triangles. As such, any collection of atoms embedded on the Eisenstein plane is always a collection of triangles, no matter their configuration. To this end, it becomes useful to define a *placement object* as a specific collection of triangles considered for placement. In this way the smallest placement object is a triangle, followed by pentagonal and hexagonal faces with five or six triangles, and the subsequent multi-triangle symmetry groups consisting of larger number of triangles that will be discussed later on.

Since every placement object is a collection of triangles, an extension of the existing placement cycle design from Figure 4.6 can be made. Therefore, any function placing an object has to be able to take a single work-arc from the workset and subsequently determine and place all the object's triangles. This can be achieved using the existing logic for the placement of single triangles, and using it on a generated set of dependent arcs from the work-arc that uniquely define all of the object's triangles. In this way, the dependent arcs (which can be viewed as work-arcs themselves) can be passed through the placement cycle of from Figure 4.12 one by one. For a hexagonal face, this implies five face-work-arcs are generated in addition to the original work-arc taken from the workset as. The process is illustrated in illustrated in Figure 4.17. Starting from the initial work-arc $[u, v]$, each subsequent face-work-arc can be found by cyclically using clockwise Eisenstein multiplication. First by multiplication on the work-arc, then on the arc found after the first multiplication, then the second, and so on until all arcs are found. Afterwards, the collection of face-work-arcs can

be passed to the triangle placement function and placed as in Figure 4.17 one at a time. The arc-array, placed-array, workset and state variables are updated per set of triangle arcs, but now for five or six sets at a time in a single placement step of the unfolding.

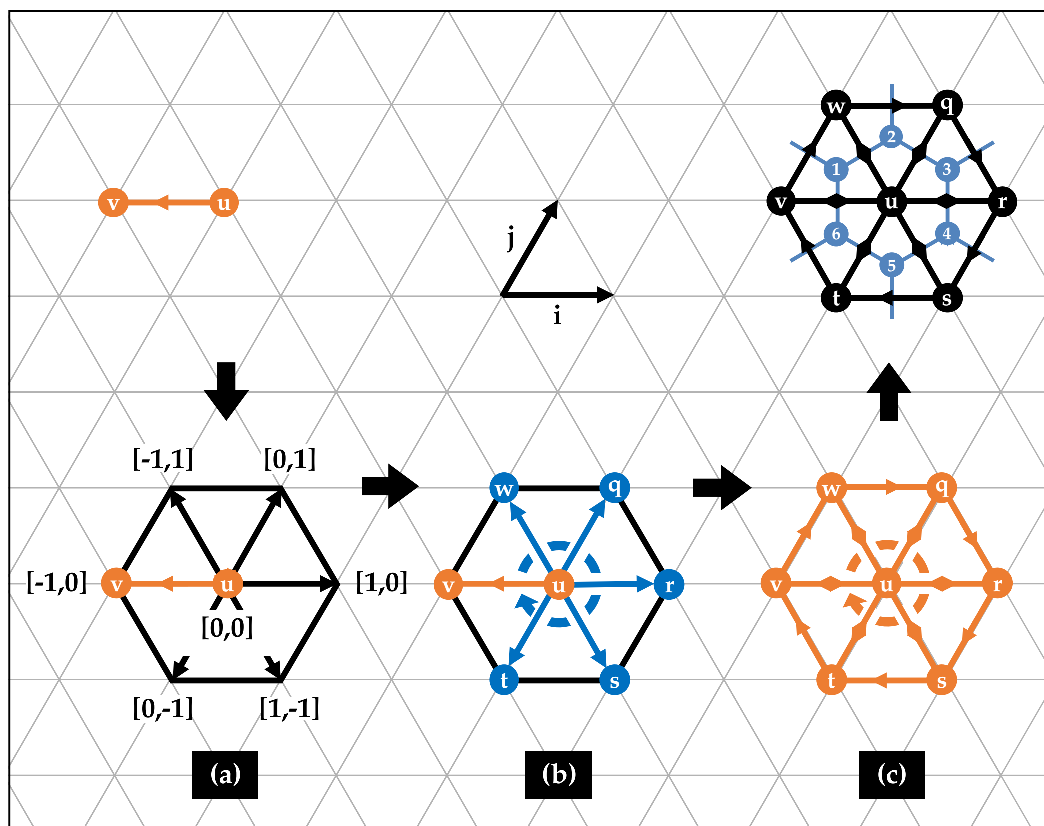


Figure 4.17: A face can be placed by **(a)** taking a work-arc $[u, v]$ from the workset and **(b)** determining all corresponding face-work-arcs (blue). **(c)** Each face-work-arc can then be used to determine and place all triangles that make up the face one by one.

After a first face is placed, the question becomes how new faces can be attached to the periphery. There appear to be two physically valid options, which are shown in Figure 4.18. As per the figure, The first option is to take the example work-arc $[b, g]$, find the third triangle node h and create a face with h at its centre. The new face consists entirely of new atoms and is connected by a single bond on the periphery with the existing molecule. The second option is to take the work-arc and build the new face around b instead. The result is that the two faces a and b share the

cubic edge $[1,6]$, which can be seen as a 'hinge connection' between the existing precursor-molecule and the newly placed face. The reason for this nomenclature is that the work from Heuser et al. (2021) revealed that for the autoassembly process precursor-molecules can be conveniently modelled as rigid faces that are allowed to move at the hinges. Notwithstanding, since both options are physically valid the algorithm should allow for both ways of connecting faces.

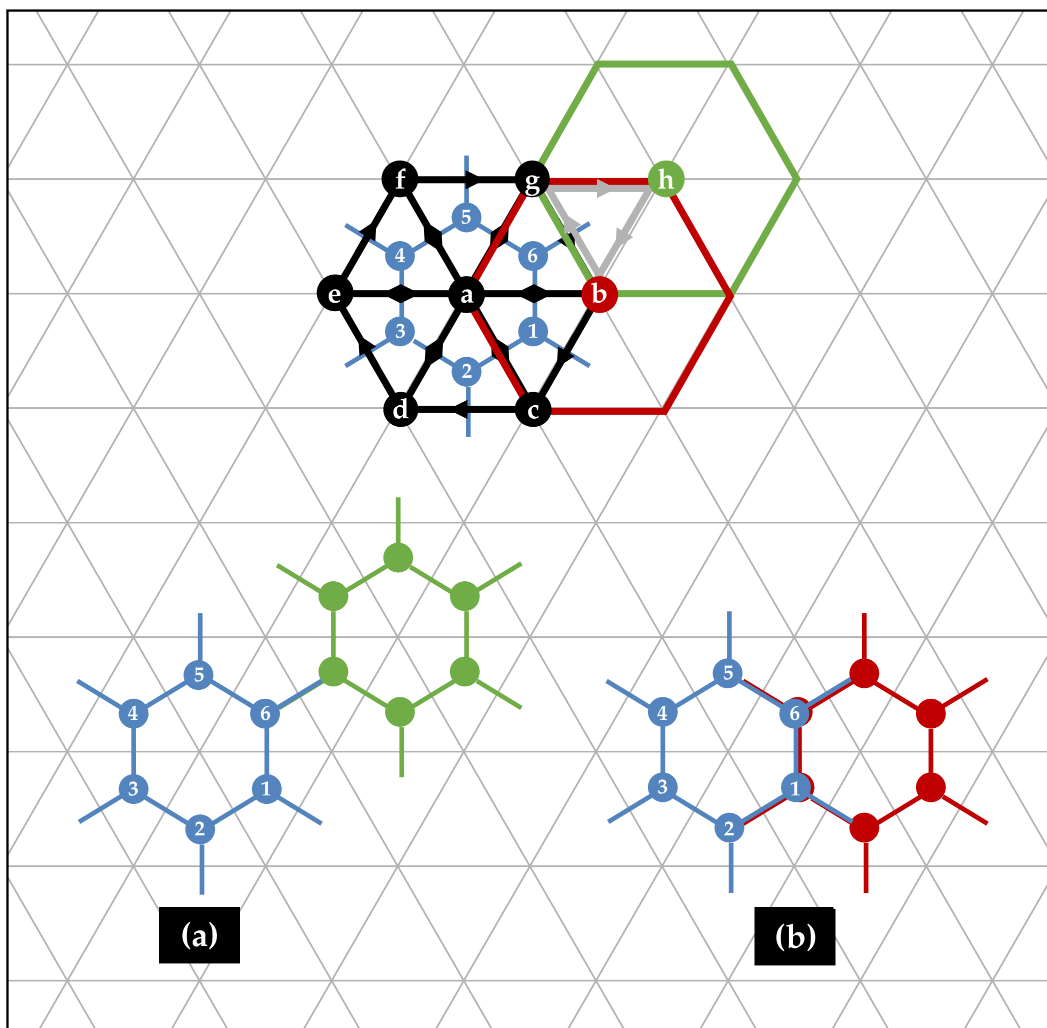


Figure 4.18: Possibilities for placing the face with work-arc $[b, g]$ using either a loose bond **(a)** or hinge **(b)** connection to the existing precursor-molecule.

When it comes to connecting faces with hinges, a caveat that arises is that the hinge triangles are part of two faces and overlap. The original constraint test would detect a collision between the occupied grid points on the plane and the intended

placement coordinates of the hinge triangles, disallowing the placement of the face. However, the overlapping triangles are completely identical in terms of their nodes and coordinates, and their placement would only overwrite existing data in the arc-array with the exact same information. As such, any overlapping identical triangles, call them *duplicate triangles*, should not stop the algorithm from placing the face.

One way to achieve this would be to explicitly identify and allow duplicate triangles to be placed, but this would clearly result in a repetition of process steps such as storing the same arcs in the arc-array again, checking their reverse for addition to the workset even though they are duplicate, and so on. A better solution would be to identify the face-work-arcs that produce the duplicate triangles and exclude them from the subsequent placement cycle of face triangles of Figure 4.17.c. One way to do so is to check for each of the face-work-arcs if the source node from the placed arc is the same as the source node from the face-work-arc and is placed in the same location. If this is the case, the face-work-arc should be excluded from the placement cycle. The adjusted process for placing faces is shown in Figure 4.19, where duplicate triangles are excluded from placement.

Naturally, if any overlapping triangle is not a duplicate and its arcs differ from the intended triangle, the collision check would still reject the placement of the face. In other words, the final constraint-function should be able to identify what obstructions are and are not part of the current placement object. If the intended placement object collides with existing objects that are not part of itself, nothing of the intended object should be placed. However, if the object collides with anything that is part of itself such as the two hinge triangles when placing faces, the object can be placed.

As it turns out, duplicate work-arcs can also make their way into the workset when singular triangles are placed and form pentagons. As an example, consider pentagon centred by node b in Figure 4.20.a with the red wedge cut-out. As part of the pentagonal face, triangle $5 : [d, b, g]$ is placed earlier than $7 : [b, e, g]$, since $[d, b]$ is queued before $[b, e]$ in the workset. After $5 : [d, b, g]$ is placed, note that $[g, b]$ actually

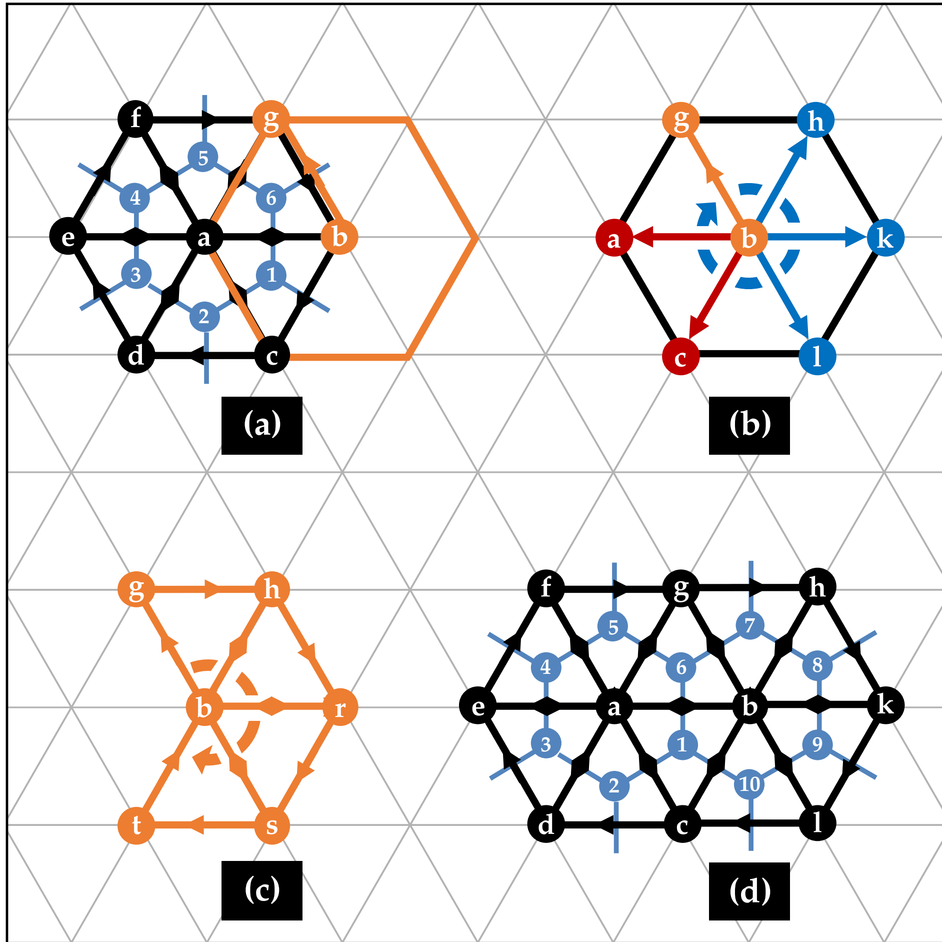


Figure 4.19: A face can be placed with a hinge connection by **(a)** taking a work-arc, **(b)** determining all corresponding face-work-arcs (blue) and excluding any duplicate triangle face-work-arcs (red). Each face-work-arc can then be used to determine and place all triangles that make up the face as seen in **(c)** and **(d)** respectively.

is added to the workset, with intended placement location in opposite direction of $[b, g]$. However, before the workset reaches the work-arc $[g, b]$, it is already placed as part of $7 : [b, e, g]$ using work-arc $[b, e]$. Note also how $[b, g]$ as a reverse arc from triangle 7 is not added to the workset because it already exists as part of triangle 5. At this point, $[g, b]$ is a duplicate arc in the workset and should not be placed. Because of this, every work-arc should first and foremost be checked against the placed-array to see if the triangle exists on the Eisenstein grid already, and reject the arc's placement object for placement if this is the case. As an additional efficiency step, reverse-arcs that are formed at each placement cycle should also be checked against the placed-array before they are added to the workset. By doing so, duplicate work-arcs are not unnecessarily considered for placement and rejected when they are taken from the workset in future placement steps.

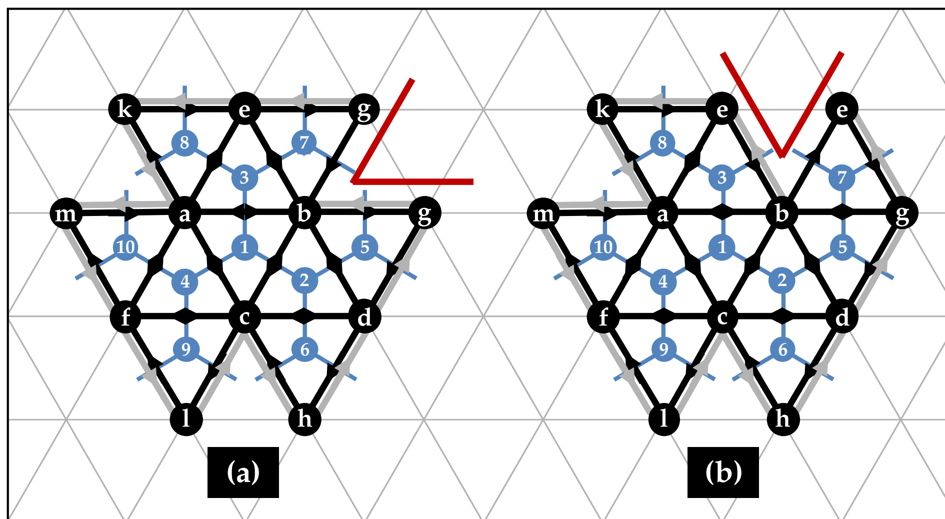


Figure 4.20: The location of the pentagon cutout is dependent on the order in which triangles are placed. **(a)** triangle $7 : [b, e, g]$ is placed, resulting in the cutout at node g. **(b)** triangle $7 : [g, b, e]$ is placed, the cutout is now at node e.

Another point to make on the placement of pentagons is that the location of the wedge cut-out is not fixed. In Figure 4.20, the cut-out is seen at different locations of pentagon b, depending on the order in which the triangles are placed. In case **(b)** in the figure, where $7 : [g, b, e]$ is placed instead of $7 : [b, e, g]$ in **(a)**, it is shown how the wedge cut-out has shifted in position, and the node e is doubled instead of node g.

With regards to recursion, the counting of triangles has to be adjusted as each placement step now needs to increment the tri-count by the number of new triangles that the newly placed face added to the unfolding. Moreover, traversing down a recursion level should update the state with the information of every placed triangle of the face, which as mentioned can simply be done for each triangle in the face separately one after the other. On a final note, the base case does not change, as a valid molecule consisting only of faces still needs to have a tri-count equal to the number of fullerene atoms N.

4.2.7 Designing P.2b: Placing symmetry groups

The second way to reduce the search space of unfoldings that was discussed it to create symmetric unfoldings. This would require a way to identify and place symmetry equivalent objects at each placement step in line with the given point group of the isomer. The planar subgroup of the full point group therefore needs to be known beforehand to allow for this identification step. This planar subgroup can be determined by computing what the largest D_n or C_n axes acting on a given root-object are, which can be done by once again using the spiral algorithm from Wirz et al. (2018) to find all the possible graph isomorphisms such that each of the applied symmetry operations (e.g. rotations and reflections) take the graph into an isomorphic graph. This means first the canonical face spiral needs to be computed, after which it can be compared to the spiral that forms from the given root-object. If they are the same spiral, the symmetry operation was an isometry and is part of the point group of the precursor-unfolding. For a central triangle, D_3 will be the largest possible group, while D_5 and D_6 are possible for a central pentagon and hexagon respectively.

Given the planar point group of the fullerene isomer, the next step is to find the symmetry equivalent arcs at each placement step. The symmetry arcs form a subset of dependent arcs, whose objects all need to be placed (if able) in the same placement step. Because the Eisenstein plane is a hexagonal mesh with six unit vectors in the Eisenstein ring, arcs that are C_n equivalent around a central root-face

are related by $6/n$ Eisenstein multiplications. In this way, C_2 arcs are oriented at three multiplications from each other and C_6 arcs at a single multiplication around the rotation axis at the barycenter of the root-object, i.e. the origin. This is exemplified for the first placement step of a C_3 isomer precursor-unfolding in Figure 4.21, where two Eisenstein multiplications repeated two times yield the symmetry equivalent arcs $(g, b) \equiv (f, e) \equiv (d, c)$. Since the periphery is chosen to be in clockwise orientation, the work-arc and its symmetry equivalent arcs are found through counterclockwise multiplication using the Eisenstein integers $(0, 1)$.

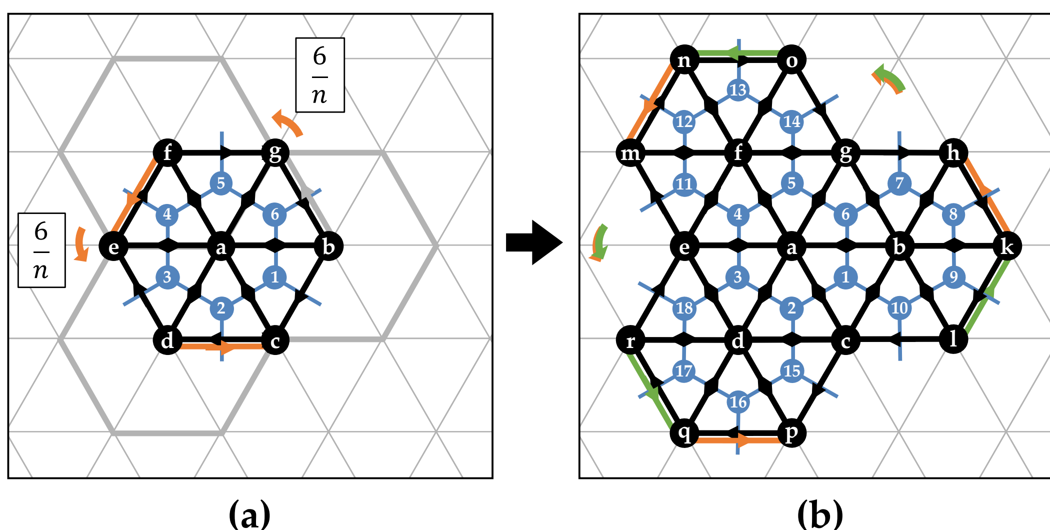


Figure 4.21: (a) C_3 symmetry equivalent arcs are found by Eisenstein multiplication of the work-arc $\frac{6}{n}$ times in counterclockwise direction $(1, 0)$, done $n - 1$ times for $n = 3$. **(b)** After placement, new sets (orange or green) of symmetry arcs can be found.

Because of the six directional symmetry of the Eisenstein plane, symmetry equivalent faces can only be placed if all of them are either pentagons or hexagons, as is depicted in Figure 4.22. It can be seen in the figure that the wedge cut-out of the pentagon breaks rotational symmetry if the other faces are hexagons. If all the faces are pentagons, the wedge cut-outs need to be oriented the same relative to the symmetry arcs as shown in the figure. If the point group moreover is of D_n and therefore includes $n \times \sigma_v$ reflection axes, the placement objects need to lie symmetric about them. Moreover, the pentagon cut-out wedges need to be oriented either perpendicular or in line with the reflection axes, as shown in Figure 4.23.

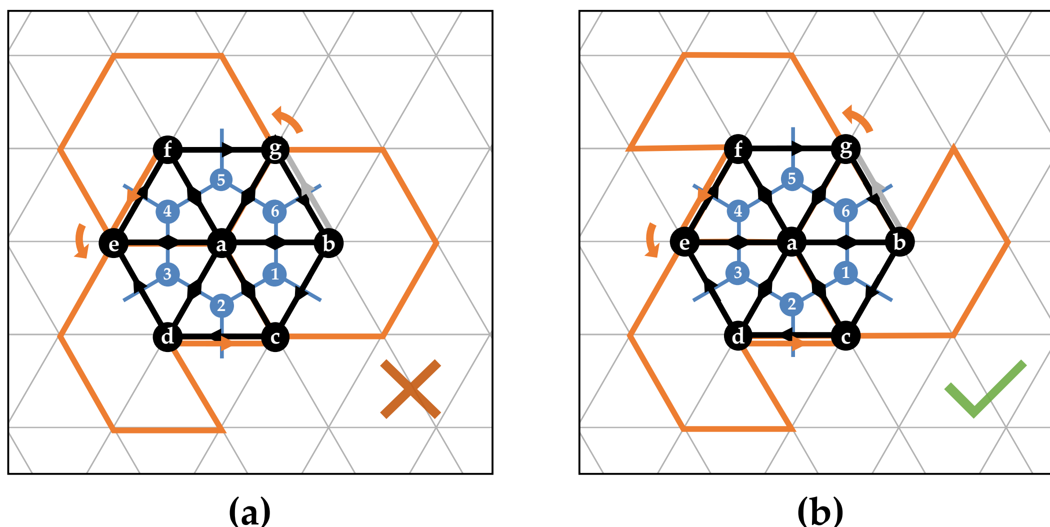


Figure 4.22: (a) Symmetry equivalent faces can only be placed if they are all either pentagons or hexagons. **(b)** If all faces are pentagons they can be placed if the cutouts have the same orientation relative to their symmetry equivalent arc.

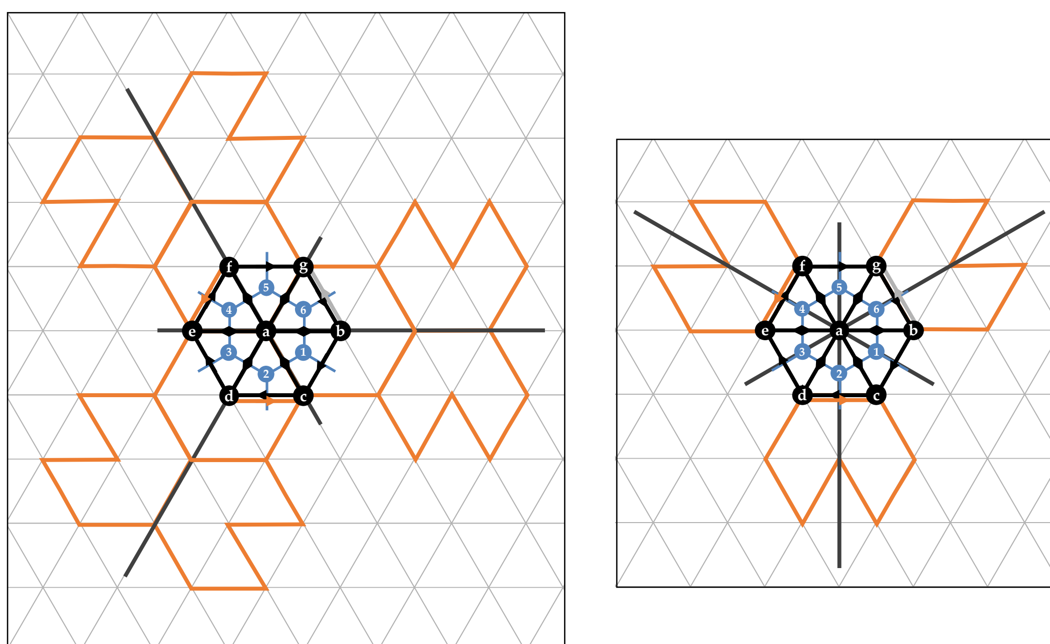


Figure 4.23: The pentagon cutouts of an isomer with point group D_n need to align with the σ_v reflection axes of the central hexagon.

An intuitive approach to finding the D_n symmetry arcs is to take the work-arc, find its rotations, and then reflect each of the C_n rotation arcs in their respective reflection axis. Or alternatively, to first reflect the work-arc in the n reflection axes,

and subsequently rotating each reflected arc exactly once. The problem with this method is that it requires a general definition of a reflection in the Eisenstein plane. In the Cartesian plane, a vector (arc) reflection across a line is given as:

$$\text{Ref}_l(v) = 2\text{Proj}_l(v) - v \tag{4.2}$$

where v is the reflected vector, l the line across which is reflected, and $\text{Proj}_l(v)$ the projection of v on the line l. However, this does not translate well to the Eisenstein plane for the reflection axes that do not align with the i and j Eisenstein axes. To circumvent this problem, it was found that it is possible to make a single reflection of the work-arc in an arbitrary reflection axis to get all the symmetry arcs. The reason for this is that all the reflected arcs are simply C_n rotations of any reflection of the work-arc. As such, a work-arc and its reflection can both be rotated $n - 1$ times using the same procedure as for C_n symmetry, resulting in $2n$ symmetric arcs that define a single D_n placement step.

The simplest axes to reflect in are the horizontal and vertical line that can be drawn through the Eisenstein plane. The horizontal reflection axis aligns with the Eisenstein direction from $(0, 0)$ to $(1, 0)$, while the vertical axis is perpendicular to that direction. In these axes, reflections can be found as:

$$\text{Ref}_{(1,0)}((i, j)) = (-(i + j), j) \tag{4.3}$$

$$\text{Ref}_{\perp(1,0)}((i, j)) = (i + j, -j) \tag{4.4}$$

where $(1, 0)$ is the horizontal axis and $\perp (1, 0)$ the vertical. It was discovered that both axes are needed to be able to get all possible symmetric unfoldings. This is visualised in Figure 4.24 for an example starting face u with D_3 symmetry. The first row depicts the difference between the axes for hinge-connected faces, where the horizontal results in three, and the vertical in six distinct faces. This difference

occurs due to the fact that the arcs in the horizontal axis have the same source nodes as their reflected counterparts, which for hinges define the same face. In the vertical axis reflections the source nodes are not the same, and therefore each arc defines a separate hinge-connected face.

For single bond connections on the other hand, the faces are found by creating the triangle of a given work-arc. This can be seen in the figure for arc (v, w) for example, which creates triangle (v, w, u), defining face u. Therefore, any reflected arcs that lie clockwise on the periphery of the unfolding define bond-connected faces that are already placed. In the figure, both the horizontal and vertical axis reflections would therefore have three arcs that define new faces, and three arcs that define the already placed centre face u. Therefore, to ensure bond-connected faces can also be placed six at a time, the reflected arcs need to be reversed as $(v, w) \to (w, v)$. The result can be seen in the bottom two depictions of the figure, where six faces can be formed using the horizontal reflection axis, and three with the vertical axis.

Because symmetry arcs can define the same face or one that is already placed, it is necessary to make sure the generated symmetry arcs all define unique and placeable faces. To this end code will need to be written that can identify and filter out any arcs that are duplicate or already placed prior to evaluating the faces for placement as would be done for non-symmetric unfoldings.

There are several things of note regarding symmetric precursor-unfoldings. The first is that for symmetric unfoldings, the choice of root-object impacts what the possible unfoldings found in the recursion space will be. The reason for this is that the symmetry group of the precursor-unfolding is dependent on the chosen root-object. A hexagon with root-node 13 may have C_3 symmetry, while hexagon 67 can yield an unfolding with D_6 symmetry, for example. This is in contrast with non-symmetric unfoldings, which will generate every possible configuration given enough computation time, making the choice of root-object arbitrary. Moreover, root-objects can be symmetry equivalent, meaning the precursor-unfolding will be the exact same regardless of the root-object. For example, for both C_{20}-I_h and C_{60}-I_h all hexagons and all atoms are symmetry equivalent. The Fullerene software package

is able to identify the symmetry inequivalent starting faces that yield different unfoldings, meaning the algorithm can disregard symmetry equivalent hexagons as root-objects.

The second thing to note is that, if the root-object of an unfolding is a face, it must always be a hexagon. This is due to the fact that 5 is a prime number, meaning pentagons can either have at least C_5 symmetry or only C_1. However, a pentagon with 5 surrounding faces will make a cone with Gaussian curvature $2\pi/6$, making the unfolding non-planar, which goes against the goals of the algorithm. If the faces are moreover placed using the hinge method as per Figure 4.18.(b), the maximal possible symmetry is of $n = 3$, because with sixfold symmetry the faces surrounding the central hexagon would collide and not be accepted for placement.

Lastly, is is important to realise that apart from faces, there are two more types of symmetry sites: triangles (atoms) and edges. Each can lead to a different symmetric unfolding, as both the rotation and reflection axes shift their positions, as for a root-triangle the rotation axis would lie at its centroid where the atom lies. In this project however, it was chosen to focus on face-symmetric unfoldings, since they are theorised to have higher chances of auto-assembly success. As such the algorithm presented in this project exhaustively searches all unfoldings with face-symmetry, but for completeness the other symmetry sites should be implemented in future work.

What is possible however, is to use the methods of creating face symmetric unfoldings and placing single symmetry equivalent triangles instead. In this way, the rotation and reflection operations for triangle arcs are the same as for those of faces, and allows the algorithm to algorithm to produce hybrid face/triangle unfoldings as well. What this means is that faces can be placed recursively until the workset in a recursion step runs empty, which would normally mean the branch in the recursion tree gets cut if the unfolding is incomplete. However, if upon unwinding to the parent node a switch would be made to placing symmetric triangles using the same workset, the unfolding might still be completed.

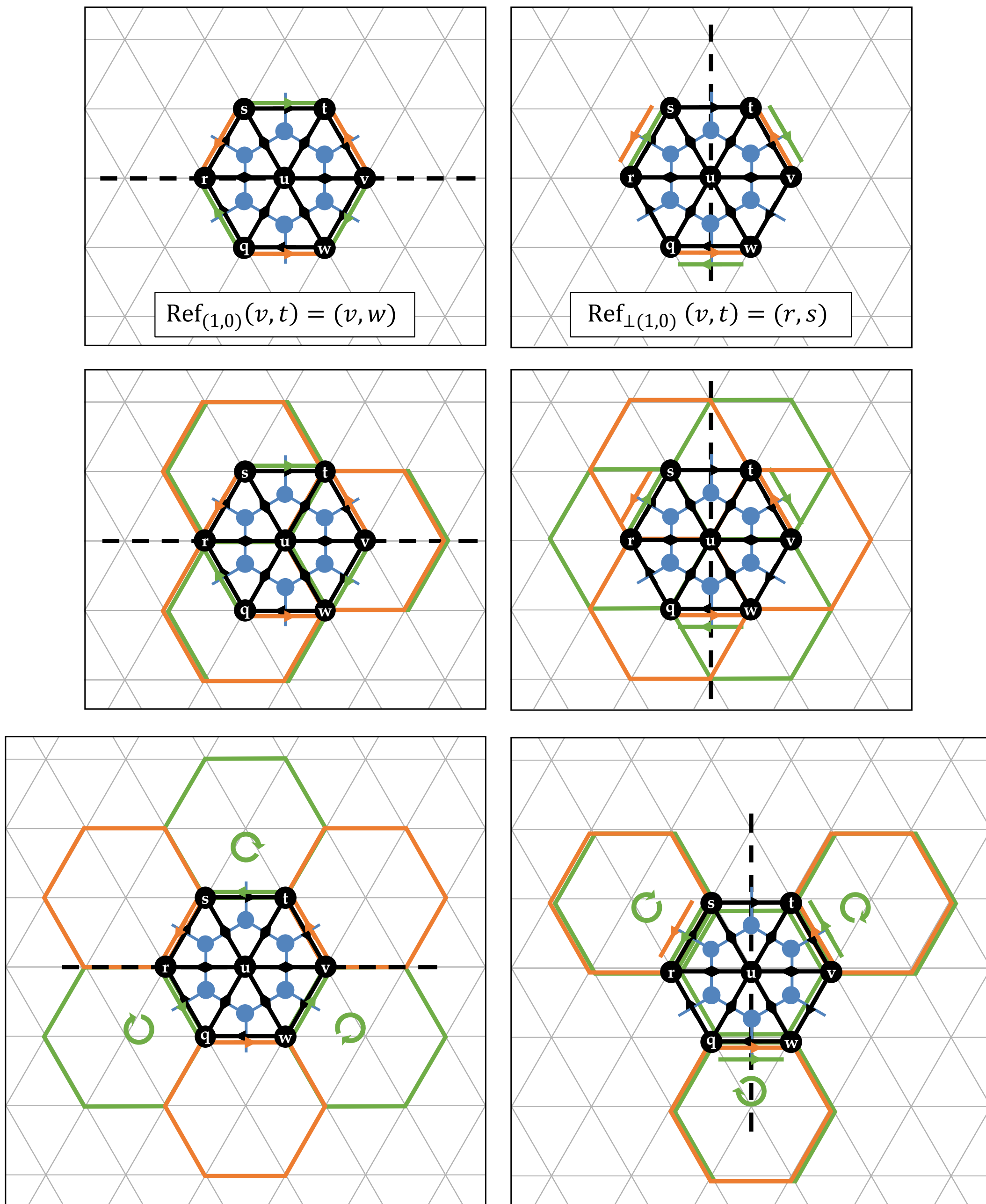


Figure 4.24: (Top) A work-arc can be reflected in the horizontal **(left)** of vertical **(right)** reflection axis. Depending on the chosen axis, the symmetry arcs that are found result in n or $2n$ faces, as exemplified for D_3 and hinge- **(middle)** and bond-connected **(bottom)** faces in the figure. For bond-connections, the reflected arcs are reversed to make sure faces can be placed six at a time as well.

4.3 IMPLEMENTATION

This section discusses how the algorithm components are implemented in terms of function definitions and variables in Python. The reason why python was chosen as the programming language due to its ease of use and readability, which are useful properties when developing a novel algorithm on a subject that has little to no reference on the matter. Moreover, the parallel masters projects from others working on CARMA related tasks are also written in Python, making it preferable to program the unfolding algorithm of this project in Python as well.

4.3.1 *Implementing P.1a: Input*

Starting with the input, the sparse adjacency matrix of the precursor-molecule bond-graph was implemented as a numpy array named `dual_neighbours`. In line with the design, the array is of size $N_f \times 6$ with its rows representing the N_f faces and the pentagons stored in the top twelve rows. Since Python only accepts rectangular arrays, the final element of each pentagonal row is assigned a dummy value -1. For example, the neighbours of face 0 centred by dual-node 0 of the C_{60}-I_h fullerene are represented in the first row of its dual-neighbour array as $[16, 15, 14, 13, 12, -1]$, which is a pentagon. The dummy value -1 was chosen because the generation of the dual-neighbour arrays are only allowed positive nodes, making it easy for any post-processing to deal with the dummy values. It is moreover preferable to have an integer dummy value because the nodes are integers as well, and single-type arrays are faster to modify and store than mixed-type arrays.

4.3.2 *Implementing P.1a: Output*

The output array that stores all the dual arcs with their Eisenstein coordinates was created as a $N_f \times 6 \times 2 \times 2$ array and was named `arc_array`. As explained in the design, it is important to realise that nodes are not stored explicitly in the array. Instead, they are implicit in the unique storage location of each arc in the array. Moreover, the 2×2 dimensions of the array are necessary to accommodate storing a

full arc with source and target coordinates as $[i_{source}, j_{source}], [i_{target}, j_{target}]$ in each element. Similar to `dual_neighbours` there are twelve dummy elements to represent the absent arcs in the pentagons, which in this case are not single integers but 2×2 sub-arrays.

The placed-array was created as a $N_f \times 6$ boolean array named `placed_array` to keep track of the placement status of all of the arcs in the unfolding. To keep this array of a single data-type as well the pentagon dummy values in the top twelve rows were set to False instead of -1. Lastly, the workset queue of unplaced arcs was constructed as a $N_{work_arcs} \times 3 \times 2$ array named `workset`, with N_{work_arcs} the number of `work_arcs` in queue for placement. The additional 3×2 dimensions store both the work-arc nodes as well as their coordinates. As discussed, this is important since the workset does not have implicit information regarding the nodes in its array structure like the `arc_array` has. It is simply a queue of arcs and moreover of variable length, so the work-arc information in each entry needs to be complete:

$$(u, v) = [[u, v], [i_u, j_u], [i_v, j_v]]$$

4.3.3 Implementing P.1a: Initialisation

In addition to the dual-neighbours array, a few other variables need to be defined to start the algorithm as shown in Code 4.1. First, the `place_objects` variable is a list of strings of the possible placement-objects. Currently, when a selection is made the unfolding will only consist of the chosen object. However, the implementation could be easily extended to allow for a variation of placement-objects at different intermediate states should the need arise. Then, if a planar point group is provided unfoldings will be generated according to it, with either triangles or faces as placement objects. If no point group is given (i.e. `point_group = 0`), all possible unfoldings in the recursion space will be generated. Next, the `constraint_function` is passed as an argument such that different generation constraints can be applied to the same unfolding process. Lastly, to limit the computation time of the algorithm a `max_unfolds` variable can be set that restricts the number of valid unfoldings

that are returned. Note that the dual-neighbours array is taken from a data-file containing dual-graph information of the chosen isomer, in this case the C_{60}-I_h fullerene.

```
1  dual_neighbours = C60ih.dual_neighbours
2  root_nodes = cf.get_root_nodes(dual_neighbours)
3  place_objects = ['triangle', 'face']
4  point_group = 'C3'
5  max_unfolds = 1
6
7  outputs = generate_unfoldings(dual_neighbours, root_nodes,
   ↪  place_objects[1], point_group, max_unfolds, cf)
```

Code 4.1: Variable initialisation and function call that start the recursive unfolding process.

Once a fullerene isomer and the unfolding parameters are defined, the function `generate_unfoldings()` takes care of producing all possible precursor-unfoldings for all possible root-objects according to the specified placement object and point group as shown in Code 4.2. For each root-node, `initialise_state()` creates the initial state and state variables, and places the root hexagon if a point group is specified. Moreover, the number of valid unfolding created is counted by `unfolding_count`, which is a variable shared between the recursion trees of each possible root hexagon. When the number of unfoldings is equal to `max_unfolds`, the recursion stops and the list of outputs is returned. The individual recursion trees are traversed by the `recursive_unfold()` function. On a side note, `cf` (short for common functions) is a file that contains all the subroutines created for `generate_unfoldings()` and `recursive_unfold`.

The `initialise_state()` function is listed in Code 4.4 and initialises both the empty arc-, placed-, and workset arrays that define a state, as well as the state-variables `tri_count` and `path`. The path variable is a list that stores the work-arcs of placed objects in the order that they were placed. Using the variable, the path in the recursion tree can be played back for debugging purposes, should any placement steps ever reveal unwanted behaviour. To allow for this, an accompanying

```python
def generate_unfoldings(dual_neighbours, root_nodes, place_obj,
    point_group, constraint_function, max_unfolds, cf):

    N_atoms = int((len(dual_neighbours.flatten()>=0)-12)/3)
    outputs = []

    for root_node in root_nodes:
        # Keep shared unfolding count between trees
        no_unfoldings = len(outputs)

        # Initialise tree, place root-object if point group
        init_state, init_state_vars =
            cf.initialise_state(dual_neighbours, root_node,
            point_group)

        # Traverse current root-object recursion tree and add
            valid unfoldings to list
        outputs += recursive_unfold(init_state,
            init_state_vars,...)

    print(f'{len(outputs)} valid unfolding(s) found.')

    return outputs
```

Code 4.2: Function that can generate all possible precursor-unfoldings given the dual-neighbours array of any fullerene isomer by traversing all recursion trees for all given root-nodes. Valid unfoldings are returned in a list named `outputs`.

function `play_path()` was also written that is able to recreate all the placement steps in the recursion tree in order. It can moreover stop at any chosen intermediate state for closer inspection.

Coming back to `initialise_state()`, the first lines of code define the required shape variables for the state arrays, as well as the source and target coordinates of the root-arc which defines the object that is placed first. The chosen origin and initial direction of the root-arc are arbitrary, provided that they both lie on the Eisenstein ring in relation to each other. For clarity, the origin was put at $[i, j] = [0, 0]$ with initial direction $[i, j] = [1, 0]$. Next, the function constructs the root-arc with its source the provided root-node by `generate_unfoldings()`. The target node of the root-arc is the first neighbour found in the data row `dual_neighbours[source]`

```
1  def play_path(state, state_vars, place_obj, point_group, cf):
2      ''' Plays back a recorded recursion path'''
3      outputs = []
4      path = state_vars[1]
5
6      for path_index, work_arc in enumerate(path):
7          state, state_vars = cf.update_state(...)
8          outputs += [[copy.deepcopy(state),
           ↪  copy.deepcopy(state_vars)]]
9
10     return outputs
```

Code 4.3: The `play_path()` function is able to playback the traversal steps through the recursion tree.

which contains the neighbours of the face centred by the root-node. The coordinates of the source are set equal to the origin, and the target coordinates to the initial direction. Then, the pair of nodes and coordinates are packed together to create the root-arc.

After the root-arc is created, the state arrays are initialised. First, the `arc_array` as the $N_f \times 6 \times 2 \times 2$ array filled with `np.nan` values as discussed in its design. Following, the `placed_array` is initialised as a boolean array with the shape of the dual-neighbours array, being $N_f \times 6$, which is the number of total arcs that have to be placed. Then, the root-arc is put into the initial workset that will be used in the first placement cycle. Then, the `collision_grid` dictionary variable is initalised with the root-arc Eisenstein coordinates. Moving on, the state variables are stored in a separate list from the state arrays for clarity. Naturally, the number of triangles is an integer that starts counting from zero and is named `tri_count`. Moreover, the `path` starts off as an empty list to which the placed work-arcs can be added while traversing the recursion tree. As a last step, the state arrays and state variables are packed in lists to make it easier to pass them as to other functions. If a point group is specified however, the root hexagon needs to be placed manually before starting the recursion process. This is due to the fact that the symmetry equivalent arcs to the root-arc are undefined, as no other arcs are placed on the plane yet. Therefore,

`initialise_state()` takes care of the initial placement step if a point group is specified.

```
def initialise_state(dual_neighbours, root_node, point_group):
    Nf, n_hex = dual_neighbours.shape[0], 6
    origin, init_direction = np.array([0,0]), np.array([1,0])

    # Construct root-arc
    u = root_node
    v = get_root_nbour(dual_neighbours, u)
    c_u, c_v = origin, init_direction
    root_arc = np.array([[u, v], c_u, c_v], dtype=int)

    # Initialise state arrays
    init_arc_array = np.full((Nf, n_hex, 2, 2),
    ↪   fill_value=np.nan)
    init_placed_array = np.zeros_like(dual_neighbours,
    ↪  dtype=bool)
    init_workset = np.array([root_arc])
    init_collision_grid = {tuple(c_u) : u, tuple(c_v) : v}

    # Initialise state variables
    tri_count = 0
    path = []

    state = [init_arc_array, init_placed_array, init_workset,
    ↪   init_collision_grid]
    state_vars = [tri_count, path]

    # Place root symmetry hexagon if point group is given
    # If no symmetry recursive_unfold() places root-object
    if point_group:
        work_arc, state[2] = state[2][0], state[2][1:]
        fd_arcs = get_face(work_arc, dual_neighbours)

        state, state_vars = update_state(fd_arcs, state,
        ↪   state_vars, 'face', 0, dual_neighbours)
        path += [work_arc]

    return state, state_vars
```

Code 4.4: The `initialise_state()` function creates the required output arrays and state variables for a precursor-unfolding.

4.3.4 Implementing P.1b: Generating all unfoldings

The main function that systematically explores the recursion tree for any given fullerene bond-graph was named `recursive_unfold()`, whose simplified code is shown in Code 4.5. First the list `outputs` is created which will contain all the collected nuts at the leaf nodes. Then the base case is evaluated, where valid unfoldings are appended and invalid ones discarded if it is True. If a leaf node is not reached yet, the recursive case traverses the recursion tree. Using the termination constraints, horizontal branches are created until either the workset is empty or the set maximum number of unfoldings is reached. For each branch, the creation of a new state (child node) starts by taking a work-arc from the workset and determining if and which triangles of the object corresponding to the work-arc can be placed. This is done using the passed `constraint_function()`, which returns a `place` boolean and all (if any) valid work-arcs. If the object can be placed, the new state and state variables are updated by `update_state()`: a function that places the object and updates the state arrays and tri-count. After placement the path is updated with the original work-arc, which is done outside of the state update since it should only add the original work-arc and not all valid ones. If the object can instead not be placed, the algorithm loops back to the termination constraint to take the next work-arc from the current workset of the state.

As can be seen in the code, the next step is to recursively call the same function that traverses the tree vertically from a level L to a level $L+1$ in the recursion tree. This is done by appending the result of the function call to the list of outputs. In this way, if a base case is reached and returns either a valid unfolding as `[[state, state_vars]]` or an empty list `[]`, this is returned to the previous recursion call at level L, carrying with it the appended unfolding in the 'bag' of outputs. If the termination constraints are reached before a base case is (i.e. the branch is cut), the function simply returns the same bag of outputs to its parent call using the last line `return outputs`. Lastly, when the algorithm is completely unwound back to the first function call and has explored all branches, the final list of outputs is returned.

It should be remarked that each recursion step is done using a copy of the original state. The reason for copying the original state is that Python uses call by reference rather than call by value. What this means is that any changes the `update_state()` function makes to a passed state are reflected outside of the function scope as well. As a consequence, if no copy would be made and the algorithm would unwind back from level $L+1$ to the same node at level L, the algorithm would wrongly start creating new branches on the basis of the updated state with a new placement object present. Therefore, the original state at level L needs to be preserved, which is thus achieved by copying it before it is changed by the state update function.

To provide additional insight on the matter, note how a `new_state` is constructed in level L and then passed to level $L+1$ recursively. Since Python uses call by reference, level $L+1$ mutates `new_state` (called state in level $L+1$). Therefore, on return to level L its `new_state` contains the updated state. When the algorithm at some point returns to the parent node at level $L-1$ from there, the `new_state` of level L no longer exists, as it is deallocated from memory when its scope at the function call of level L is exited. However, if the workset is not empty yet in level L (i.e. more children can be created), the `new_state` is overwritten with a new `new_state` object and passed to the next sibling at level $L+1$. At this point, the old `new_state` object is marked for garbage collection in memory and is automatically deallocated. In short, `new_state` is allocated inside the function body of any `recursive_unfold()` call and its scope is the function body. While it is passed down with another recursion call and winding up it exists, and as the algorithm unwinds and returns it is removed from existence.

```
1  def recursive_unfold(state, state_vars, N_atoms, place_obj,
   ↪  point_group, constraint_function, max_unfolds,
   ↪  dual_neighbours, cf):
2      outputs = []  # List of valid unfoldings
3
4      ## BASE CASE
5      if tri_count == N_atoms:
6          if is_valid_unfolding(state):
7              # Collect valid leaf nodes in outputs
8              return [[state, state_vars]]
9          else:
10             # Discard invalid leaf nodes
11             return []
12
13     ## RECURSIVE CASE
14     else:
15         # Horizontal branch creation
16         while workset != empty and len(outputs) < max_unfolds:
17             # Take work-arc from workset
18             work_arc, state[2] = state[2][0], state[2][1:]
19
20             # Determine which arcs/triangles can be placed
21             # valid_arcs can be multiple, singular or none
22             place, valid_arcs = constraint_function(work_arc,
               ↪  ...)
23
24             if place:
25                 # deepcopy the state for recursion because Python
                   ↪  passes by reference
26                 new_state, new_state_vars =
                   ↪  cf.update_state(valid_arcs,
                   ↪  copy.deepcopy(state, state_vars), ...)
27                 # Record path
28                 path += [work_arc]
29                 # Vertically traverse the tree (wind-up)
30                 outputs += recursive_unfold(new_state,
                   ↪  new_state_vars, ...)
31
32     return outputs
```

Code 4.5: Simplified code for the main function that recursively generates all valid unfoldings.

4.3.5 *Implementing P.1a: Unfolding constraints*

To determine whether an object can be placed the `is_placeable()` function was created, which is the function that is currently passed as the constraint function to `recursive_unfold()`. The function takes a work-arc and state as arguments and returns a `place` boolean together with all the valid object-work-arcs. To see how the function works, consider the code from Code 4.6. First, it is checked whether the work-arc is stored in the arc-array and therefore placed on the Eisenstein plane already. To this end a sub-function `is_placed()` finds the truth value of the arc in the placed-array and returns it. Shown in Code 4.7, the function `get_pos()` determines the storage location as $[\text{row}, \text{column}]$ of the arc in the placed-array. It does so by the method illustrated in Figure 4.25 for the example arc $[0, 13]$ of the C_{60}-I_h fullerene. Remember that the dual-neighbours array is of shape $N_f \times 6$ where each row represents the face that is centred by the node whose number is equal to the index of the row. As seen the figure, row 0 stores the neighbours of node 0 that together form the arcs defining face 0 of the precursor-molecule. Subsequently, since the arc $[0, 13]$ is found at found at row 0, column 3 of dual-neighbours, its corresponding truth value is also found at row 0, column 3 of the placed-array, which `is_placed()` returns. If the work-arc exists on the Eisenstein plane the function returns a boolean of 0 and no valid arcs to reject the placement object. If the work-arc does not exist yet, the function triggers the execution of the appropriate constraint function specific for each placement object.

```
1  def is_placeable(arc, state, place_obj, dual_neighbours):
2      if is_placed(arc, state[1], dual_neighbours):
3          return 0, []  # don't place, no valid_arcs
4      else:
5          if place_obj == 'triangle':
6              return tri_is_placeable(arc, state, dual_neighbours)
7
8          elif place_obj == 'face':
9              return face_is_placeable(arc, state, dual_neighbours)
10
11         elif place_obj == 'symmetry_group':
12             return symmetry_group_is_placeable(arc, state,
               ↪  dual_neighbours)
```

Code 4.6: The triangle placement constraint function returns all valid work-arcs together with a boolean that indicates whether the object can be placed.

```
1  def is_placed(arc, placed_array, dual_neighbours):
2      pos = get_pos(arc[0,0], arc[0,1], dual_neighbours)
3      return placed_array[pos[0]][pos[1]]  # True or False
```

Code 4.7: Function that finds the location of an arc boolean in the placed-array and returns its placement status.

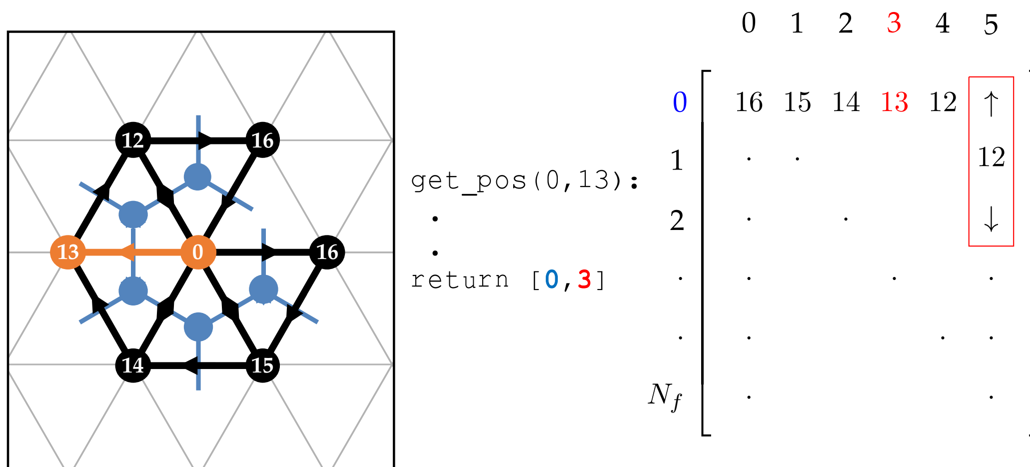


Figure 4.25: The row and column of an arc in the `arc_array` are determined by `get_pos()`.

4.3.6 *Implementing P.1a: Triangle constraints*

If the passed placement object is a single triangle, `tri_is_placeable()` takes a work-arc, determines the triangle using the `get_tri()` function shown in Code 4.8, and sees if the triangle shows any collision with the existing precursor-molecule. A collision is detected if any of the intended triangle coordinates are in the collision-grid and correspond to a node that is different from the intended one. If there is no collision that means either the placement location is completely free, or that any nodes that do exist are part of the intended triangle as well. Notwithstanding, this does not allow for duplicate triangles since any duplicate arcs are rejected at the initial `is_placed()` check, before any particular constraint-function is called. The code for the triangle constraint and collision check functions is listed in Code 4.9 and Code 4.10 respectively. Again, what is returned by `is_placeable()` is a boolean of 1 together with triangle work-arc, or a value of 0 and no work-arc.

```
1  def get_tri(arc, dual_neighbours):
2      u, v, cs_u, cs_v = arc[0,0], arc[0,1], arc[1], arc[2]
3      w = tri_next_node(u,v,dual_neighbours)
4      cs_w = tri_next_cs(cs_u, cs_v)
5
6      return np.array([u,v,w]), np.array([cs_u, cs_v, cs_w])
```

Code 4.8: Function that retrieves the third triangle node with its coordinates and returns the full triangle.

```
1  def is_collision(arc, collision_grid, dual_neighbours):
2      nodes, coords = get_tri(arc, dual_neighbours)
3
4      return any([tuple(coord) in collision_grid and
       ↪  collision_grid.get(tuple(coord))!=node for node, coord in
       ↪  zip(nodes, coords)])
```

Code 4.9: Check for collisions of a triangle intended for placement with the occupied Eisenstein grid points

```
1 def tri_is_placeable(arc, state, dual_neighbours):
2     placed_array, collision_grid = state[1], state[3]
3
4     return (not is_placed(arc, placed_array , dual_neighbours)
      ↪  and not is_collision(arc, collision_grid,
      ↪  dual_neighbours)), arc
```

Code 4.10: Function that determines whether a triangle can be placed.

4.3.7 Implementing P.1a: Face constraints

Moving on to constraining the placement of faces, the `face_is_placeable()` function was developed to determine the set of face-work-arcs that can be placed and is listed in Code 4.12. As discussed in the design, any work-arcs that would form duplicate triangles on the Eisenstein plane are excluded from the array of valid arcs. Before this is done, the `get_face()` function finds all face-work-arcs cyclically using the passed work-arc and the dual-neighbours array. Because this process is non-trivial, pseudo-code instead of the full code of the function is listed in Code 4.11. Essentially, the method entails looping through the row of neighbours of the source node of the given work-arc in the dual-neighbours array, determining their coordinates using cyclical Eisenstein multiplication, and making sure the data structure of the the face-work-arcs is correct for further processing. This results in an array of arcs with each element $[[u, v], [i_u, j_u], [i_v, j_v]]$.

From the array of all face-work-arcs, the exclusion of duplicate triangle work-arcs is done by first checking if any of the face-work-arcs are already placed. If the face-work-arc exists on the plane already, it is subsequently checked whether the coordinates of the source node from the placed-arc are the same as the source coordinates from the face-work-arc. In this way, if the coordinates are the same the face-work-arc would produce a duplicate triangle and is thus marked for exclusion from the array of face-work-arcs. After all face-work-arcs are gone through, the arcs marked for exclusion are deleted. Following, the remaining face-work-arcs are evaluated for placement as ordinary triangles using `tri_is_placeable()`. As such, if any of the remaining face-work-arcs would place a triangle that is either

already placed or would yield a collision, the entire face is rejected. If not, the valid face-work-arcs are returned together with the boolean 1 that states the face can be placed.

```
1  def get_face(arc, dual_neighbours):
2      # Start with face center node = work-arc source node
3
4      ## Form all node pairs as
5      # [center node, neighbour 1], [center node, neighbour 2] ...
       ↪  etc.
6      # Creates e.g. [[0,16],[0,15],[0,14],[0,13],[0,12]] for first
       ↪  pentagon of C60-ih
7
8      ## Form all coordinate pairs
9      # Starting with the given work-arc, find all coordinate pairs
10     # [center node [i,j], neighbour 1 [i,j]] etc..
11
12     # Reshape into work-arc structure: [[n1,n2],[i_n1,
       ↪  j_n1],[i_n2,j_n2]] for all arcs
13
14     return face_work_arcs
```

Code 4.11: Simplified code for the function that finds all face-work-arcs of a pentagonal or hexagonal face given a single work-arc.

```
def face_is_placeable(arc, state, dual_neighbours):
    fwas = get_face(arc, dual_neighbours)  # face_work_arcs

    del_indices = []

    for index, fwa in enumerate(fwas):

        if is_placed(fwa, placed_array, dual_neighbours):
            pos = get_pos(fwa[0,0], fwa[0,1], dual_neighbours)

            pa_source_coords = arc_array[pos[0]][pos[1]][0]
            fwa_source_coords = fwa[1]

            if np.array_equal(fwa_source_coords,
            ↪   pa_source_coords):
                del_indices.append(index)

    fwas = np.delete(fwas, del_indices, axis=0)

    place = all([tri_is_placeable(fwa, placed_array,
    ↪   collision_grid, dual_neighbours)[0] for fwa in fwas])

    return place, fwas
```

Code 4.12: Constraint function for faces that removes duplicate triangles from the placement cycle and returns whether to place the face together with the remaining valid face-work-arcs.

4.3.8 *Implementing P.1a: Symmetry group constraints*

Symmetry equivalent arcs can only be placed if all their corresponding placement objects can be placed simultaneously. Moreover, if the placement objects are faces all of them have to be either pentagons or hexagons. To this end, the `sym_is_placeable()` function was created to find the symmetry equivalent arcs of a given work-arc and determine whether they can be placed on the Eisenstein plane. The code is listed in Code 4.16, and starts by obtaining all the symmetry equivalent arcs to the work-arc using `get_sym()` as shown in Code 4.13.

The `get_sym()` function finds the arcs by considering what the point group and placement object are, and executing the appropriate code. As shown in Code 4.14,

for C_n symmetrical faces the $n-1$ symmetry arcs are found by using $6/n$ counter-clockwise Eisenstein multiplication steps of the work-arc to find the coordinates. Then, the nodes can be found by making use of the fact that their reverses are already placed on the Eisenstein plane. As such, its nodes are stored in the collision-grid and correspond to the nodes of the symmetry arc as well. If the given point group is D_n instead as shown in Code 4.15, the work-arc is reflected using `easy_reflect()`, which uses equations Equation (4.3) and Equation (4.4) to determine the coordinates of the reflection, and the collision-grid to find the nodes. The work-arc and reflected arc are then rotated $n-1$ times using the same code as for C_n symmetry, yielding $2n$ symmetry arcs that are ready to be evaluated for placement.

```
1  def get_sym(arc, collision_grid, dual_neighbours, place_obj,
   ↪   face_con, point_group, ref_axis):
2      clockwise = [1,-1]
3      counterclockwise = [0,1]
4
5      group = point_group[0]
6      n = int(point_group[1])
7      mult_steps = int(6/n)
8
9      if group == 'C':
10         # See Cn part of get_sym() code snippet
11
12     elif group == 'D':
13         # See Dn part of get_sym() code snippet
14
15     return sym_arcs
```

Code 4.13: Simplified code for the function that finds all symmetry equivalent arcs given a work-arc and planar point group of the precursor-unfolding.

Once the symmetry arcs are found, the function `sym_is_placeable()` was created to determine all the dependent arcs of the objects corresponding to the symmetry arcs, and see if they can all be placed. The structure of the function is shown in Code 4.16, with separate snippets for C_n and D_n faces listed in Code 4.17 and Code 4.19 respectively.

```python
'''Cn part of the get_sym() function'''
# Initialise sym_arcs with work_arc
sym_arcs = [arc]

mult_steps = int(6/n)

if place_obj == 'face':
    source, target = arc[0][0], arc[0][1]
    source_cs, target_cs = arc[1], arc[2]

    # Because there are n-1 new rotationally equivalent sym_arcs to determine
    for i in range(n-1):
        # Do 6/n Eisenstein multiplications to get to next rotation sym_arc
        for i in range(mult_steps):
            next_source_cs = eis_mult(source_cs, counterclockwise)
            next_target_cs = eis_mult(target_cs, counterclockwise)
            source_cs = next_source_cs
            target_cs = next_target_cs

        next_source = collision_grid.get(tuple(next_source_cs))
        next_target = collision_grid.get(tuple(next_target_cs))

        next_rotation_arc = [[next_source, next_target], next_source_cs, next_target_cs]
        sym_arcs += [next_rotation_arc]

return sym_arcs
```

Code 4.14: Snippet of code that finds all C_n equivalent symmetry arcs.

For symmetrically equivalent faces there are several more constraints and edge-cases that show up. First of all, the faces corresponding to the symmetry arcs can only be placed if they are either all pentagons or all hexagons. To this end the total number of arcs that will be considered is made by running `get_face()` on all the symmetry arcs. If the number of arcs for each face of each symmetry arc is the same, that implies the faces are all pentagons or hexagons, and the function can proceed. Next, the dependent arcs are found for each of the symmetry faces using `face_is_placeable()`, which removes the dependent arcs that form

```
'''Dn part of the get_sym() function'''
# Reverse needed to ensure 'bond' can place 6 faces at a time
# If reverse would be False for 'bond' then reflection can get
↪  invalid [None, node] arcs.
if (face_con == 'bond'):
    reverse=True
else:
    reverse=False

# Reflect work_arc and initialise sym_arcs with the work_arc and
↪  its reflection
reflection = easy_reflect(...)
reflect_arcs = [arc, reflection]
sym_arcs = copy.copy(reflect_arcs)

if place_obj == 'face':
    # Get all rotations for the work_arc and its reflection
    for ref_arc in reflect_arcs:
        source, target = ref_arc[0][0], ref_arc[0][1]
        source_cs, target_cs = ref_arc[1], ref_arc[2]

        # Get all n-1 rotation sym_arcs
        # Same as Cn, see Cn_get_sym
        ...

return sym_arcs
```

Code 4.15: Snippet of code that finds all D_n equivalent symmetry arcs.

the hinge and evaluates each of the triangles of the face-dependent-arcs using `tri_is_placeable()`.

There are several edge cases that can occur, the one that is shared between C_n and D_n is that symmetry faces can 'steal' triangles from one another. For example, if two symmetry faces a and b share triangles but are placed in separate locations, the arcs of the shared triangles will only be placed as part of whichever face is placed last. The reason for this is that `face_is_placeable()` evaluates each of the symmetry faces in a vacuum, without considering the placement of the other symmetry faces. Therefore, the function `is_interim_placed()` was created, which takes the generated dependent arcs and runs them through an interim placement cycle. The function returns *True* if the placement of any of the triangles corresponding to the

dependent arcs interfere with the placement of any of the other triangles. In this way, it is prevented that triangles are re-placed in different locations when they are 'stolen' from another object. Together with `face_is_placeable()`, it was found that these constraints are enough to yield valid C_n unfoldings.

```
1  def sym_is_placeable(arc, state, state_vars, dual_neighbours,
   ↪ place_obj, face_con, point_group, ref_axis):
2      # Get arcs that define the symmetry equivalent objects
3      sym_arcs = get_sym(...)
4
5      # For triangles, the dependent dep_arcs are equivalent to the
       ↪ sym_arcs
6      if place_obj == 'triangle':
7          return all(tri_is_placeable(...), sym_arcs
8
9      # For faces, the dep_arcs can be found using
       ↪ face_is_placeable for each sym_arc
10     elif place_obj == 'face':
11
12         if group == 'C':
13             # See Cn part of sym_is_placeable() code snippet
14
15         elif group == 'D':
16             # See Dn part of sym_is_placeable() code snippet
17
18     return place, dep_arcs
```

Code 4.16: Simplified code for function that takes a work-arc and determines if the objects formed by all symmetry equivalent arcs can be placed and returns all their dependent arcs.

For D_n unfoldings, there are several more considerations to be made. As was shown previous in Figure 4.24, reflected arcs are already placed when they lie clockwise on the periphery, or duplicate if the (reversed) reflection is the same arc as its counterpart. For this reason, a function `filter_sym_arcs()` was created to filter out any unwanted arcs that should not be evaluated for placement. After all, should they be kept, both `is_interim_placed()` and `face_is_placeable()` could potentially flag the whole placement step as invalid for the wrong reasons. The code is shown in Code 4.20, and shows that arcs are removed if they share a face-center (source node) or are duplicates (placed and in exact location of the symmetry arc).

```
'''Cn part of the sym_is_placeable() function'''
# If sym faces are all pentagons or all hexagons
for arc in sym_arcs:
    total_arcs += len(get_face(arc, dual_neighbours, face_con))

arcs_per_face = (total_arcs/len(sym_arcs))

if arcs_per_face == 5 or arcs_per_face == 6:
    # Find all face dependent arcs of all symmetric faces
    for sym_arc in sym_arcs:
        dep_arcs += face_is_placeable(...)[1]

    # 1. All faces have to be placeable.
    # 2. is_interim_placed() returns True if sym_faces are
    ↪  'stealing' triangles from each other
    # By checking if every dep_arc triangle can be placed ...
    # ...with knowledge of the other triangles being placed
    if all(face_is_placeable(...)[0] for arc in sym_arcs) and not
    ↪  is_interim_placed(...):
        return 1, dep_arcs
    else:
        return 0, []

# If not all faces are pentagons or hexagons
else:
    return 0, []
```

Code 4.17: Snippet of code that finds and evaluates all C_n equivalent dependent arcs for placement.

```
def is_interim_placed(dep_arcs, state, state_vars,
↪  dual_neighbours):
    s, sv = copy.deepcopy(state), copy.deepcopy(state_vars)
    for dep_arc in dep_arcs:
        if tri_is_placeable(dep_arc, s, dual_neighbours)[0]:
            s, sv = place_tri(dep_arc, s, sv, dual_neighbours)
        else:
            return 1
    return 0
```

Code 4.18: Function that evaluates symmetry objects by means of an interim placement cycle. If any of the dependent arcs of an object would render the placement of any of the other objects invalid, the symmetry objects cannot be placed.

Moreover, any arcs that occur more than once in the list are removed as well by `remove_double_arcs()`, to ensure faces are not placed twice, which result in the wrongful double counting of new triangles.

Coming back to Code 4.19, there are additional considerations to be made if the faces are pentagons. For one, pentagons cannot be placed D_n symmetric in groups of ≤ 3 for hinge-connections. This is due to the fact that it is impossible to align three or less hinge-connected pentagon cut-outs along D_n axes, which was also visualised in Figure 4.22.b and Figure 4.23. It is possible to do so if the faces are placed with bond-connections instead, but only if the pentagon cut-outs are shifted to align with the axes. Therefore a function `shift_cutout(dep_arcs, n=2)` was created that changes the order of the dependent arcs generated by `face_is_placeable()` as shown in Figure 4.26. In the standard case on the left of the figure, the pentagon cut-out is always made after the last dependent arc as generated by `get_face()`. Therefore, by shifting arcs 2 and 3 in clockwise direction on the Eisenstein ring, the cut-out will be right before the nth dependent arc which correctly aligns it with the reflection axis. If any number of pentagons ≥ 3 with either hinge- or bond-connections are placed, the location of the cutout is arbitrary, as its reflection will always point in the opposite direction of the axis ensuring correct D_n symmetry.

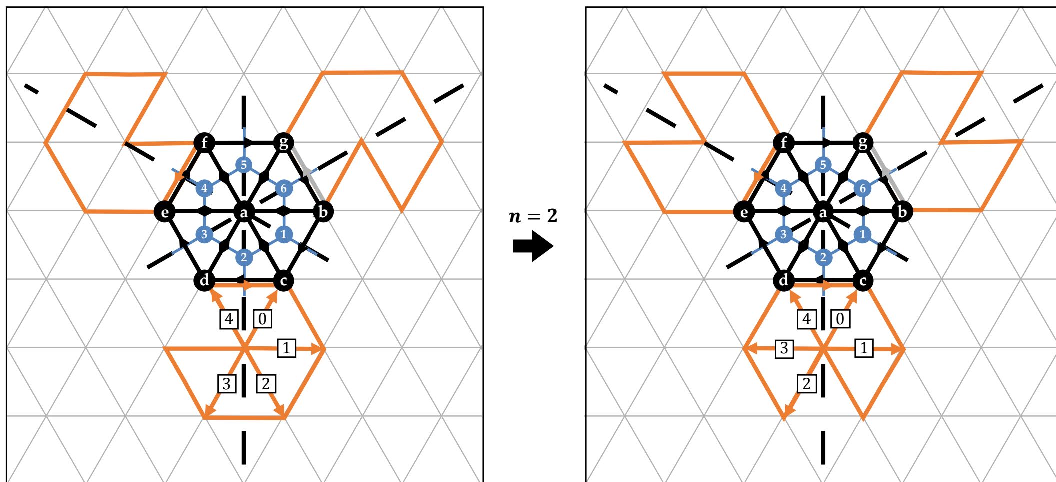


Figure 4.26: For D_n symmetric unfoldings that try to place ≤ 3 pentagons in a placement step, the pentagon cutout needs to be shift to align with the reflection axes by rotating arcs n and $n+1$ on the Eisenstein ring using a single Eisenstein multiplication in clockwise direction.

For symmetric triangles, the logic for finding the symmetry arcs is exactly the same as for faces, again using $n - 1$ times $6/n$ Eisenstein rotations for C_n, and creating their reflections for D_n in the horizontal and vertical axes. When it comes to the constrains for placement, the only difference is that for triangles, the symmetry arcs already are the dependent arcs, and can therefore be directly evaluated using `tri_is_placeable()`. Similarly, for D_n symmetry the arcs are filtered using `filter_sym_arcs()` and run through `is_interim_placed()` to ensure no triangles are allowed to be re-placed.

```
'''Dn part of the sym_is_placeable() function'''
# make_unique_sym_arcs() removes any sym_arcs that define the
↪ same face ...
# ... to prevent that placement get blocked by
↪ is_interim_placed()
sym_arcs = make_unique_sym_arcs(...)

for arc in sym_arcs:
    total_arcs += len(get_face(...))

arcs_per_face = (total_arcs/len(sym_arcs))

# Sym faces need to be either all pentagons or all hexagons
if arcs_per_face == 5 or arcs_per_face == 6:
    if arcs_per_face == 5:
        # Hinge pentagons can only be placed 6 at a time...
        # ...as the cutouts of <= 3 pentagons cannot be Dn
        if len(sym_arcs) <= 3:
            if face_con == 'hinge':
                return 0, []

            # Bond pentagons can be placed in threes but only if
            ↪ you shift the cutout
            elif face_con == 'bond':
                for sym_arc in sym_arcs:
                    face_dep_arcs = face_is_placeable(...)[1]
                    face_dep_arcs = shift_cutout(face_dep_arcs)
                    dep_arcs += face_dep_arcs

    # Otherwise, cutout location is arbitrary
    else:
        for sym_arc in sym_arcs:
            dep_arcs += face_is_placeable(sym_arc, state,
            ↪ state_vars, dual_neighbours, face_con)[1]

    # 1. All faces have to be placeable.
    # 2. is_interim_placed() returns True if sym_faces are
    ↪ 'stealing' triangles from each other
    # By checking if every dep_arc triangle can be placed ...
    # ...with knowledge of the other triangles being placed
    if all(face_is_placeable(...)[0] for arc in sym_arcs) and not
    ↪ is_interim_placed(dep_arcs, state, state_vars,
    ↪ dual_neighbours):
        return 1, dep_arcs
    else:
        return 0, []

# If not all faces are pentagons or hexagons
else:
    return 0, []
```

Code 4.19: Snippet of code that finds and evaluates all D_n equivalent dependent arcs for placement.

```
1  def filter_sym_arcs(dual_neighbours, sym_arcs, state, place_obj,
   ↪  face_con):
2      arc_array, placed_array = state[0], state[1]
3      unique_sym_arcs = []
4
5      if place_obj == 'face':
6          face_centers=[]
7
8          for sym_arc in sym_arcs:
9              face_center = sym_arc[0][0]
10
11             if face_center not in face_centers and not
               ↪  is_duplicate(dual_neighbours, sym_arc, arc_array,
               ↪  place_obj):
12                 unique_sym_arcs += [sym_arc]
13                 face_centers += [face_center]
14
15     elif place_obj == 'triangle':
16         for sym_arc in sym_arcs:
17             if not is_duplicate(dual_neighbours, sym_arc,
               ↪  arc_array, place_obj):
18                 unique_sym_arcs += [sym_arc]
19
20     # Double arcs are unplaced sym_arcs that occur twice in the
       ↪  generated list of sym_arcs
21     # Double arcs are blocked by interim triangle placement and
       ↪  therefore have to be removed beforehand
22     unique_sym_arcs = remove_double_arcs(unique_sym_arcs)
23
24     return unique_sym_arcs
```

Code 4.20: Function used in `sym_is_placeable()` to remove symmetry arcs from the list that would result in invalid an placement step or the needless constraining thereof.

4.3.9 Implementing P.1a: Hybrid unfoldings

To create (symmetric) hybrid face-triangle unfoldings, an extension was written to the `recursive_unfold()` function as listed in Code 4.21. The code switches to triangles as placement objects once a dead end in the recursion tree is reached, which is the case when no objects were placed at a recursion level, meaning no children nodes were created from a certain parent node. This is checked by setting a boolean `dead_end` to *True* upon entry to a recursive step, and switching it to false if any placement steps are made while the workset is not empty. Once all workset options are exhausted and nothing is placed, `dead_end` will remain *True*, at which point the algorithm will try to close out the unfolding with (symmetric) triangles recursively. An additional parameter `threshold` was defined as a percentage of completeness unfoldings must have before they are hybridised. What this means is that by setting a threshold of 0.9, only unfoldings that are at least 90% complete are finished with triangles, while a threshold of 0 means every dead end is pursued with hybridisation.

Since it is possible that arcs were removed from the workset due to the fact that its face(s) were unplaceable, the workset needs to be replenished with the entire unplaced periphery of the unfolding at the dead end. The reason for this is that invalid arcs for face placement are not necessarily invalid for triangles as well. To this end, a function `get_periphery()` was created as shown in Code 4.22. The function takes the current output unfolding, finds all the arcs that are unplaced using a function `get_unplaced_arcs()` from the placed-array, and cross-checks them with the arc-array to see if its reversed counterpart exists. If so, the unplaced arc is added to the list of periphery arcs which will be used as the new workset for the hybrid recursive calls.

Because it is possible to reach dead ends after switching placement object as well, a safeguard was implemented to make sure the algorithm would not attempt to recurse further after all the options for triangle placement are exhausted. This

was done by introducing a list variable called `place_priorities`, which contains the placement objects in the order that they should be tried. In this way, if the placement priorities are `['face', 'triangle']`, the algorithm will place faces until a dead end is reached, at which point it will switch to triangles as they are next in the list by removing the current placement object and passing `place_priorities[1:]` to the recursive function. As such, placement priorities that contain a single object such as `['face']` will not attempt to create hybrid unfoldings, but only allow the placement of full faces. Using the strategy of placement priorities should also make it easier to extend the possible placement objects in the future, should the need arise.

```
1  '''Hybrid unfoldings in recursive_unfold()'''
2  # Creates hybrid unfoldings if the algorithm is having trouble
   ↪  closing out unfoldings with only faces
3  # Reconstruct the periphery to use as workset, since unplaceable
   ↪  work-arcs for faces might be placeable for triangles
4  if dead_end and len(place_priorities) > 1:
5      workset = cf.get_periphery(dual_neighbours, [state,
       ↪  state_vars])
6
7      if threshold:
8          if tri_count >= int(threshold*N_atoms):
9              tree_outputs += recursive_unfold(...)
10     else:
11         tree_outputs += recursive_unfold(...)
12
13 return tree_outputs
```

Code 4.21: To ensure face-symmetric unfoldings can be completed, an extension was created for `recursive_unfold` that continues recursively with `place_obj = 'triangle'` once a dead end is reached in the tree, with the possibility to set a threshold to govern how complete unfoldings must be before they are hybridised.

```
1  def get_periphery(dual_neighbours, output):
2      state = output[0]
3      arc_array = state[0]
4      placed_array = state[1]
5      periphery = []
6
7      unplaced_arcs = get_unplaced_arcs(dual_neighbours,
       ↪  placed_array)
8
9      for arc in unplaced_arcs:
10         pos = get_pos(dual_neighbours, arc[1], arc[0])
11         source_coords = arc_array[pos[0]][pos[1]][0].tolist()
12         target_coords = arc_array[pos[0]][pos[1]][1].tolist()
13
14         if not any(np.isnan(source_coords)):
15             periphery += [[arc, [int(x) for x in target_coords],
               ↪  [int(x) for x in source_coords]]]
16
17     return periphery
```

Code 4.22: Function that gets the periphery arcs of an unfolding by finding unplaced arcs and comparing them to their reverses in the arc-array. Used by `recursive_unfold()` to replenish the workset when a dead end is reached and hybrid unfoldings are allowed.

4.3.10 Implementing P.1a: Updating the unfolding

Once it is determined that an object can be placed, the `update_state()` function was created to take care of placing the object and updating the state arrays and variables accordingly. The function code is shown in Code 4.23, where it can be seen that the all work-arcs valid for placement determined by the constraint-function are passed directly to the function. Similar to the constraint-function `is_placeable()`, `update_state()` is essentially implemented as a switch that triggers different placing methods depending on the passed placement object. In this way, when the placement object string 'triangle' is passed a single triangle is placed and the `tri_count` is incremented by one, whilst the string 'face' would place five or six triangles and update the count with five or six as well. All placement functions called by `update_state()` return the state arrays that now contain the new placement-object information, as well as the incremented tri-count.

```
def update_state(valid_arcs, state, state_vars, place_obj,
    dual_neighbours):
    if place_obj == 'triangle':
        return place_tri(valid_arcs, state, state_vars,
            dual_neighbours)

    elif place_obj == 'face':
        return place_face(valid_arcs, state, state_vars,
            dual_neighbours)

    elif place_obj == 'symmetry_group':
        return place_symmetry_group(valid_arcs, state,
            state_vars, dual_neighbours)
```

Code 4.23: The `update_state()` function stores new data in the state arrays and updates the tri-count.

4.3.11 *Implementing P.1a: Placing triangles*

First on the list of possible objects is the triangle. To place triangles the `place_tri()` function was created, which takes a work-arc and updates all the state arrays with the triangle arcs, placement status, and any newly found work-arcs. Shown in Code 4.24, the method starts by determining the nodes and coordinates that form the triangle using `get_tri()` which was discussed before and is shown in Code 4.8.

The two main operations that follow are to place the triangle on the Eisenstein plane and to update the workset with any newly found peripheral work-arcs. First, the triangle is reversed:

$$[[a, b, c], [[a_i, a_j], [b_i, b_j], [c_i, c_j]]] \Rightarrow [[c, b, a], [[c_i, c_j], [b_i, b_j], [a_i, a_j]]] \,, \tag{4.5}$$

which is done outside the loop because the reversing array operation can be easily done using $[:: -1]$. Then, the function loops through the three nodes and coordinates of the determined triangle. In this way, each pair of arc coordinates is first stored in their unique location in the arc-array, which is found using `get_pos()`. Then, the placement status of the arc is switched to True in the placed-array. Afterwards, its reverse-arc $[u, v]$ is formed and checked against the `placed_array` to determine

whether it is already placed. The placement status of each arc can once again be found using `get_pos()` to find the location in the placed-array. If the value is True, the arc is already placed and hence discarded, while a value of `False` means the reverse-arc isn't placed yet and needs to be added to the workset. The process is then repeated for the remaining two triangle arcs, storing their coordinates and potentially adding their reverse to the workset.

In the last part of the function, the (coordinate : node) pair of the newly found third triangle node is added to the collision-grid as it now occupies a grid-point on the Eisenstein plane. Moreover, the tri-count is incremented by one to reflect the placement of the new triangle, after which the state is packed up again and returned back to the `recursive_unfold()` function as the `new_state` and `new_state_vars`.

It was chosen to combine updating the workset and placing the triangle in a single function because the workset is always updated after a triangle is placed, with no exceptions. Therefore, separation of the two would serve no purpose and only introduce room for error should one be unintentionally executed without the other.

```python
def place_tri(arc, state, state_vars, dual_neighbours):
    # Unpack state for readability
    arc_array, placed_array, workset, collision_grid = state

    nodes, coords = get_tri(arc, dual_neighbours)

    # Add third triangle node to collision_grid
    collision_grid[tuple(coords[-1])] = nodes[-1]
    state_vars[0] += 1  # tri_count += 1

    # Get reverse triangle for workset update
    rev_nodes = nodes[::-1]
    rev_coords = coords[::-1]

    for i in range(3):
        # Place triangle
        pos = get_pos(nodes[i], nodes[(i+1) % 3],
        ↪  dual_neighbours)
        arc_array[pos[0]][pos[1]] = [coords[i], coords[(i+1) %
        ↪  3]]
        placed_array[pos[0]][pos[1]] = True

        # Update workset
        u, v = rev_nodes[i], rev_nodes[(i+1) % 3]
        cs_u, cs_v = rev_coords[i], rev_coords[(i+1) % 3]
        r_pos = get_pos(u, v, dual_neighbours)

        # If not placed add reverse-arc to workset
        if (placed_array[r_pos[0]][r_pos[1]] == False):
            work_arc = np.array([[[u, v], cs_u, cs_v]])
            workset = np.concatenate((workset, work_arc))

    # Pack up and return
    state = [arc_array, placed_array, workset, collision_grid]
    return state, state_vars
```

Code 4.24: Function that places single triangles by updating the state arrays and state variable tri-count.

4.3.12 *Implementing P.2a: Placing faces*

Using the existing logic of the `place_tri()` function, placing faces is straightforward. The code is shown in Code 4.25, where it can be seen that each of the face-work-arcs determined by `face_is_placeable()` are simply passed

to `place_tri()` until all valid face triangles are placed. Doing so does introduce several duplicate work-arcs to the workset, namely a copy of each of the face-work-arcs. The reason for this is that when a triangle is placed as part of the face, `place_tri()` does not have placement information on all the triangles that will follow. In this way, each face-work-arc places a triangle with one of its arcs the reverse of the next face-work-arc in the queue. And of course, when `place_tri()` then reverses this reverse of the face-work-arc, it sees it is not placed and ends up adding an exact copy of the face-work-arc to the workset. Because a placement check is done for each work-arc in the `is_placeable()` function however, these duplicates are automatically rejected. In terms of efficiency, it was therefore not deemed worthwhile to prevent their addition to the workset.

Placing faces naturally updates more elements in the state arrays and increments the tri-count for each new triangle that is placed as part of the face. Note for example how a whole hexagonal face stores 18 unique arcs, with 15 arcs for the pentagon, and changes an equal amount of values to True in the `placed_array`. Moreover, the workset is updated with any unplaced reverse arcs from the face periphery as well. Moreover, a group of face-work-arcs fills up a whole row of the `arc_array`, as they are all the neighbour arcs stemming from the source centre node of the face.

```
1 def place_face(face_work_arcs, state, state_vars,
  ↪ dual_neighbours):
2     # face_is_placeable() has determined all valid face_work_arcs
      ↪ for placement
3     # And whether face is placeable at all
4     for face_work_arc in face_work_arcs:
5         state, state_vars = place_tri(face_work_arc, state,
          ↪ state_vars, dual_neighbours)
6
7     return state, state_vars
```

Code 4.25: A face is placed by placing the triangles corresponding to its valid face-work-arcs one by one using the triangle placement function.

4.3.13 *Implementing P.2b: Placing symmetry groups*

Similar to `place_face()`, symmetry equivalent objects can be placed by simply looping over all of the symmetry equivalent and possibly face dependent arcs and passing them to `place_tri()` one by one. In practice, the created `place_sym()` function has the same code functionality as `place_face()`. Notwithstanding, the two are separated for clarity of the functioning of the algorithm.

```
1 def place_sym(dep_arcs, state, state_vars, dual_neighbours):
2     for arc in dep_arcs:
3         state, state_vars = place_tri(arc, state, state_vars,
          ↪  dual_neighbours)
4
5     return state, state_vars
```

Code 4.26: Place all symmetry dependent arc triangles and update the state and state_vars accordingly.

4.3.14 *Plotting unfoldings on the Eisenstein plane*

The completed algorithm returns a list of valid precursor-unfoldings, where each is unfolded according to the specified object and symmetry parameters. Practically, the unfoldings are arc-arrays containing all the coordinate information of each of the unfolding arcs. To plot the dual arcs and cubic atoms on the Eisenstein plane, the plotting function `plot_unfolding()` was created. The code is shown in Code 4.27, where the function takes an arc-array and plots the dual graph arcs, cubic graph atom nodes, and equilateral Eisenstein plane. Since negative coordinates can be assigned to arcs during placement, the dummy values in the arc-array are `np.nan` instead of -1, as they can be filtered out using NumPy's `np.isnan()`.

The Eisenstein integer coordinates can be plotted in Cartesian space by using a conversion matrix M as written in line 7 of Code 4.27. The converted coordinates can then be plotted using `ax.plot(xs,ys)`, where `xs` and `ys` are the x and y coordinates for the source and target node of the arc respectively, which is then repeated for each arc in the array. Since each row of the arc-array stores a face of

the unfolding, the row indices can be used to determine the node number to be plotted. In this way, going through the first row plots the arcs between node 0 and its five or six neighbours, and numbers the center node 0 using `ax.text()`. While `ax.plot()` draws the edges between the nodes, the directional arrows are plotted on the midpoint of the edge using `ax.quiver()`. Then, the atom is plotted together with its half bond perpendicular to the plotted arc in clockwise direction, meaning the atom is plotted 'inward' at the center of the triangle. The result of plotting a single entry and an entire row in the arc-array is shown on the left and right of Figure 4.27 respectively. After all arcs, atoms, and bonds are embedded on the plane, the triangular grid is drawn using `ax.triplot()`. Lastly, the x and y axis ticks and are converted to Eisenstein integers, and adjusted dynamically based on the size of the unfolding to ensure it fits.

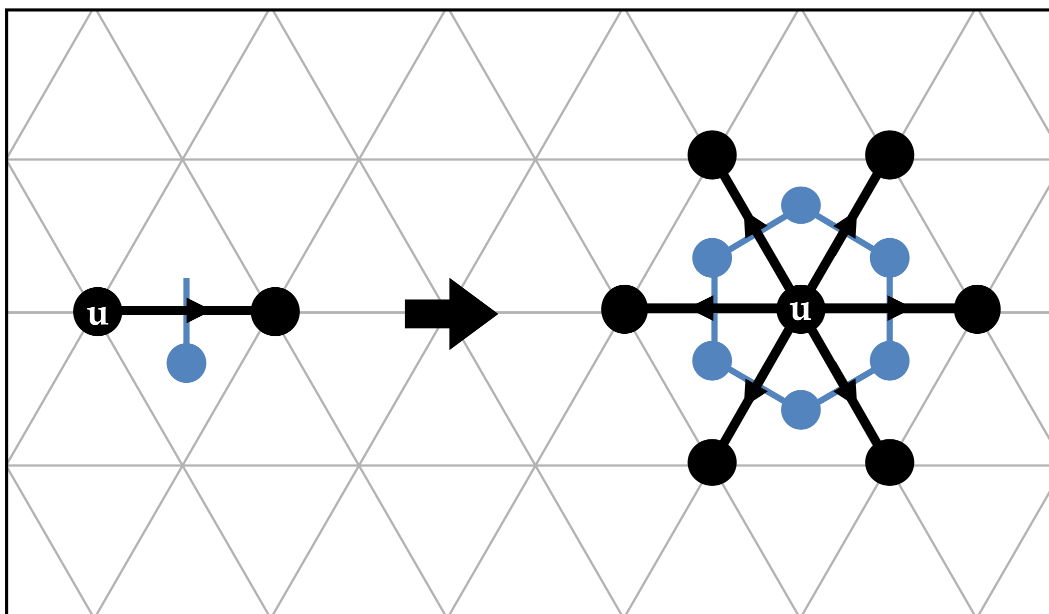


Figure 4.27: A single unfolding plotting step embeds an arc together with its orthogonal half bond and atom **(left)**. Plotting an entire row of the arc-array results in six face arcs whose triangles will be completed by the neighbouring faces.

```
def plot_unfolding(arc_array, fig, ax, plot_scale=1, title='',
    legend=True, lims=[]):
    ## Eisenstein to Cartesian conversion matrix
    M = np.array([[1, np.cos((2*np.pi)/6)], [0,
        np.sin((2*np.pi)/6)]])

    ## Plot all arcs
    for face_num, face in enumerate(arc_array):
        for arc_cs in face:
            # Don't plot nan valued arcs from arc_array
            if np.isnan(arc_cs).any():
                pass
            else:
                # Convert Eisenstein to Cartesian
                start, end = M.dot(arc_cs.T).T

                # Plot edge i.e. two connected nodes
                ax.plot(xs, ys, ...)

                # Plot node number on node that numbers face
                ax.text(xs[0], ys[0], ...)

                # Plot edge direction to make it an arc
                ax.quiver(pos_x, pos_y, dx/norm, dy/norm, ...)

                # Plot cubic graph atom and half bond clockwise
                    normal to the arc: nxs, nys
                ax.plot(nxs, nys, ...) # bond
                ax.plot(nxs[1], nys[1], ...) # atom at source of
                    bond

    ## FIGURE STYLING
    # Find smallest and largest values of i, j in the arc_array
        to use as limits
    # Ticks are at converted Eisenstein to Cartesian values
    # Set labels to Eisenstein integers, not Cartesian

    ## Plot Eisenstein grid
    # Create box to triangulate using the calculated limits
    # Create grid points and triangulate them
    grid_tris = triplt.Triangulation(grid_x, grid_y)
    ax.triplot(grid_tris, ...)
```

Code 4.27: Simplified code for the `plot_unfolding()` function, which embeds an arc-array of a completed unfolding onto a drawn Eisenstein plane.

5 RESULTS AND DISCUSSION

5.1 RESULTS

5.1.1 *Unfoldings*

The constructed algorithm generates precursor-unfoldings according to a set of specified parameters: the isomer, (planar) point group, placement priorities, face connection type (hinge or bond), reflection axis (horizontal or vertical), and symmetry inequivalent root-node (starting hexagon). Because the purpose of this project is to create a correct and functional algorithm, a selection of seven isomers of varying sizes and point groups were chosen. The parameters are listed in Table 5.1, and make for a total of $28 \times n_{\text{root}}$ combinations, where n_{root} is the number of root-nodes for a particular isomer. Therefore, the total number of recursion trees that were generated is equal to $28 \times 25 = 700$. Three dimensional models of the chosen isomers were generated using the *PyMol* software package, and are shown in Figure 5.1 for visual reference.

Isomers	C_{60}-I_h	C_{120}-D_6	C_{120}-D_{6d}	C_{120}-T_d	C_{140}-I_h	C_{180}-I_h	C_{540}-I_h
No. root-nodes	1	5	5	5	1	3	5
Place objects	face, triangle, hybrid						
Face connections	hinge, bond						
Point groups	C_1, C_2, C_3, C_6, D_2, D_3, D_6						

Table 5.1: The parameters used to generate unfoldings with the algorithm according to all possible combinations.

To evaluate the algorithm, three distinct runs were made: First, the algorithm was given thirty minutes to generate non-hybrid unfoldings with either faces or triangles, which was done in combination with each of the other possible parameters. The run went on until the time limit was reached or a single valid unfolding was found. For the second run, the placement priority was set to `['face', 'triangle']` with a threshold of 0.65 (65% complete) to create hybrid unfoldings for all combinations that did not yield a valid unfolding within the time limit of 30 minutes with a new time limit of 10 minutes. The third run was made to evaluate the variety of unfolding paths the algorithm generates by running two distinct parameter combinations with `max_unfolds = 150` of the C_{540}-I_h isomer. The three runs will hereof be referred to as the 'discovery run', the 'hybrid run', and the 'variety run'.

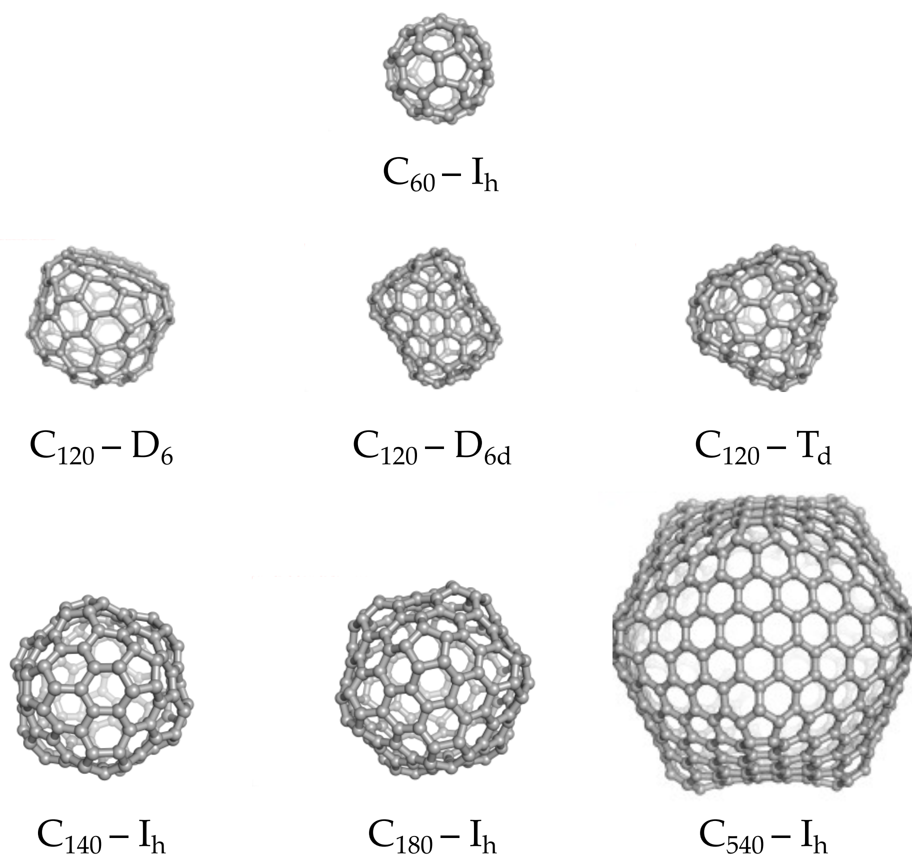


Figure 5.1: The isomers for which unfoldings were generated in this project as visualised using the *PyMol* software package.

The results of the discovery and hybrid runs are tabulated in Table 5.3 for face and hybrid placement, and Table 5.2 for triangle placement. In the tables, a value of '1' indicates a valid unfolding was found, 'y' a valid hybrid unfolding, '-' if neither was found within the time limit, and '0' if the search space was exhausted before the time limit was reached. Starting with Table 5.3, it should be noted that the root-node at which the valid unfolding was found is not listed, as the purpose of the discovery run was to find any valid unfolding for the isomer given the symmetry and placement object constraints. Out of the 119 combinations, 10 non-hybrid and 24 hybrid solutions were found, the time limit was reached in 45 cases, and for the remaining 40 cases the search space was exhausted. It can be seen that valid non-symmetric C_1 unfoldings were found for all isomers hinge-connected and bond-connected faces, as well as for C_3 hinge-connected faces. Moreover, no valid D_6 face unfoldings were found for any isomer with the exception of C_{540}-I_h , where the time limit was reached in three out of the four D_6 cases.

The valid unfoldings of each parameter combination were written to python files together with all the information and parameters of the run and shall be called the 'output file'. A second file that was continuously updated with the unfolding output that has the highest number of placed triangles was moreover created, and will be referred to as the 'dump file'. By creating a dump file, it is possible to inspect how far the algorithm was able to complete the unfolding, as well as inspect the recursion path to that unfolding.

In the figures on the following pages, collections of plotted unfoldings are presented for three of the seven isomers, namely the small C_{60}-I_h , the medium-sized C_{120}-D_6 , and the large C_{540}-I_h . The unfoldings are ordered by their point group (row) and face connection type (column) if applicable. Unfoldings were marked with a 'y' if it is a hybrid, and also denote the number of triangles that were placed. If the threshold of 65% was not reached for a parameter combination, the deepest level of the first recursion path was taken to plot for further inspection. The collections of unfoldings are followed by a close-up of two valid and two invalid unfoldings to be

able to assess the validity of the unfoldings in greater detail. Additional unfoldings for each of the remaining isomers are shown in Appendix A for completeness.

Moving on to the variety run, 150 unfoldings with a threshold of 60% were generated for each of the five root-nodes of C_{540}-I_h using non-hybrid C_2 bond-connected faces. The objective of this run is to compare the influence of the starting hexagon on the shape of the unfolding as well as examine the variety in unfolding paths the algorithm takes. Secondly, a D_2 run was done for each root-node using triangles and hybrid hinge- and bond-connected faces. This second run was made to identify if the influence on the shape of the unfolding by the root-node can is shared across the different placement objects. The isomers and parameter combinations chosen for the variety run were done based on the notion that the larger C_{540}-I_h fullerene should have a wider variety of unfolding paths, making it more likely to get an interesting divergence between them.

From each of the 150 unfoldings four unfoldings were selected and plotted, as can be seen in Figure 5.10. The figure lists the root-nodes on the left-most column, and the unfolding number in the top row. For the second run, the results are depicted in Figure 5.11 and ordered for each of the root-nodes and placement objects.

Isomer	C_1	C_2	C_3	C_6	D_2 h	D_2 v	D_3 h	D_3 v	D_6 h	D_6 v
C_{60}-I_h	1	-	1	-	-	-	-	0	0	0
C_{120}-D_6	1	1	1	1	1	-	-	-	-	-
C_{120}-D_{6d}	1	1	1	1	-	-	-	-	-	-
C_{120}-T_d	1	1	1	-	-	-	-	-	-	-
C_{140}-I_h	1	1	1	-	-	-	-	-	-	-
C_{180}-I_h	1	1	1	-	-	-	-	-	-	-
C_{540}-I_h	1	1	1	-	-	-	-	-	-	-

Table 5.2: Results of the discovery run for triangle unfoldings, with a value 1 if valid unfoldings were found, 0 if not, and '-' if none were found within the time limit. The abbreviations h and v denote the horizontal and vertical reflection axes respectively.

	C_1		C_2		C_3		C_6		D_2				D_3				D_6			
Isomer	g	b	g	b	g	b	g	b	g_h	g_v	b_h	b_v	g_h	g_v	b_h	b_v	g_h	g_v	b_h	b_v
C_{60}-I_h	1	y	0	-	1	h	0	0	0	0	0	0	0	0	0	0	0	0	0	0
C_{120}-D_6	1	y	y	y	1	y	y	y	-	-	-	-	0	-	-	-	0	0	0	0
C_{120}-D_{6d}	1	y	y	y	1	y	1	0	-	-	-	-	0	0	0	0	0	0	0	0
C_{120}-T_d	1	y	y	y	y	y	y	0	-	-	-	-	0	0	-	-	0	0	0	0
C_{140}-I_h	1	y	-	0	-	-	-	-	0	0	-	-	0	0	0	0	0	0	0	0
C_{180}-I_h	1	y	-	-	-	-	0	0	0	0	-	-	0	0	0	0	0	0	0	0
C_{540}-I_h	1	y	-	-	y	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0

Table 5.3: Results of the discovery run for face and hybrid unfoldings together with 3D-models of the isomers. The entries have a value of 1 if valid unfoldings were found, 0 if not, 'y' for a hybrid unfolding, and '-' if none were found within the time limit. The parameter abbreviations are hinge (g), bond (b), and subscripts h and v to indicate the horizontal and vertical reflection axes respectively.

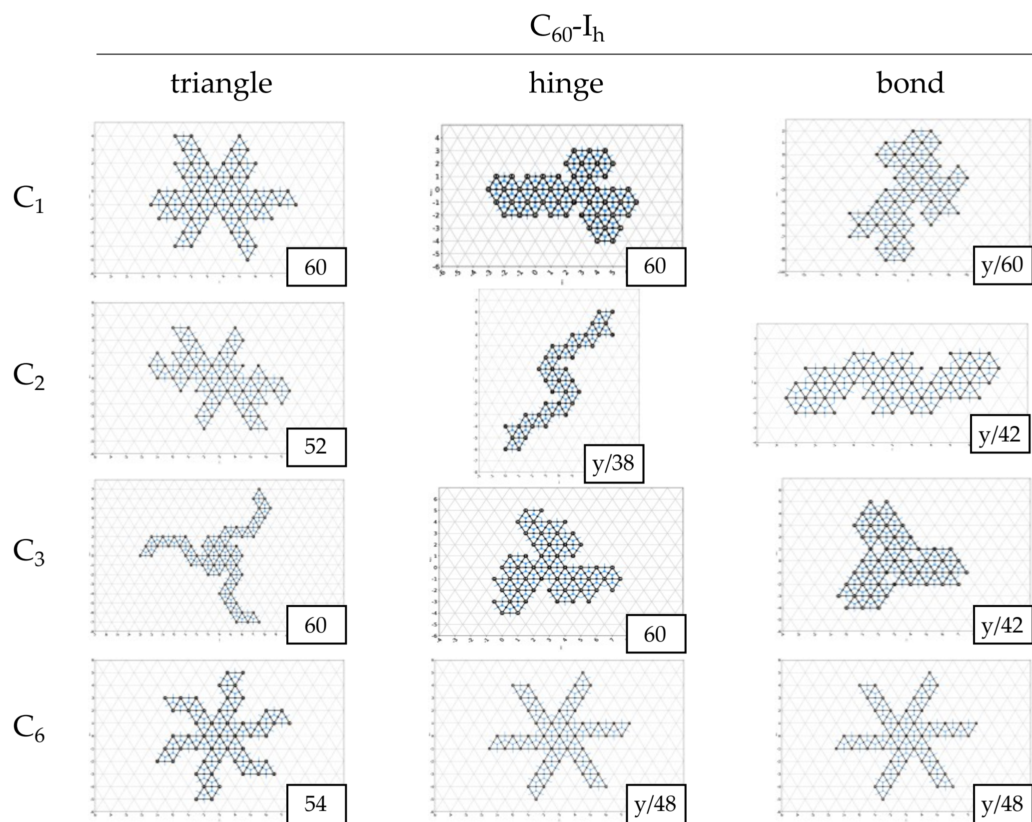


Figure 5.2: Collection of C_n unfoldings generated in the discovery run for C_{60}-I_h , marked with a 'y' for hybrid unfoldings and the number of triangles placed.

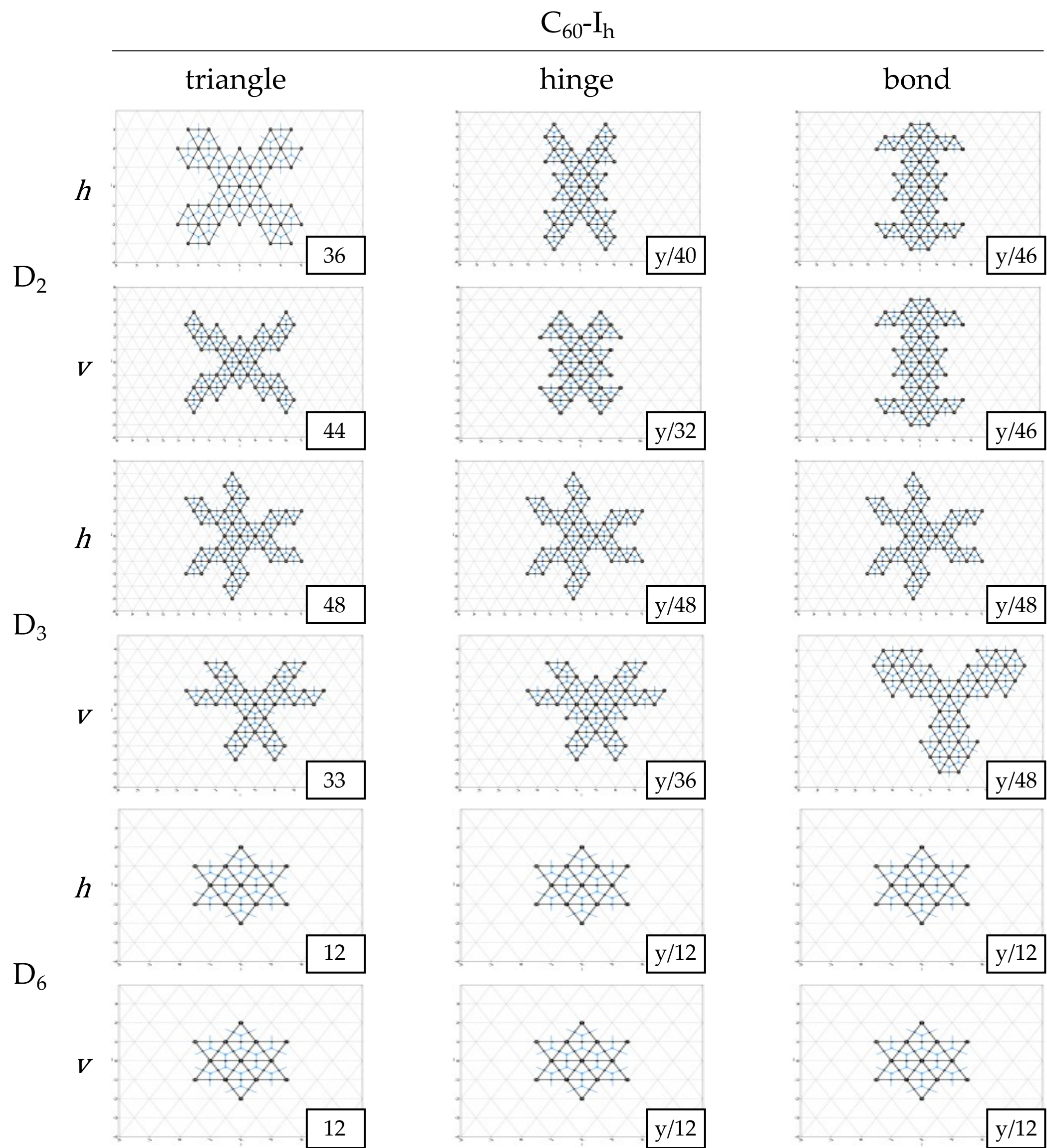


Figure 5.3: Collection of D_n unfoldings generated in the discovery run for C_{60}-I_h , marked with a 'y' for hybrid unfoldings and the number of triangles placed.

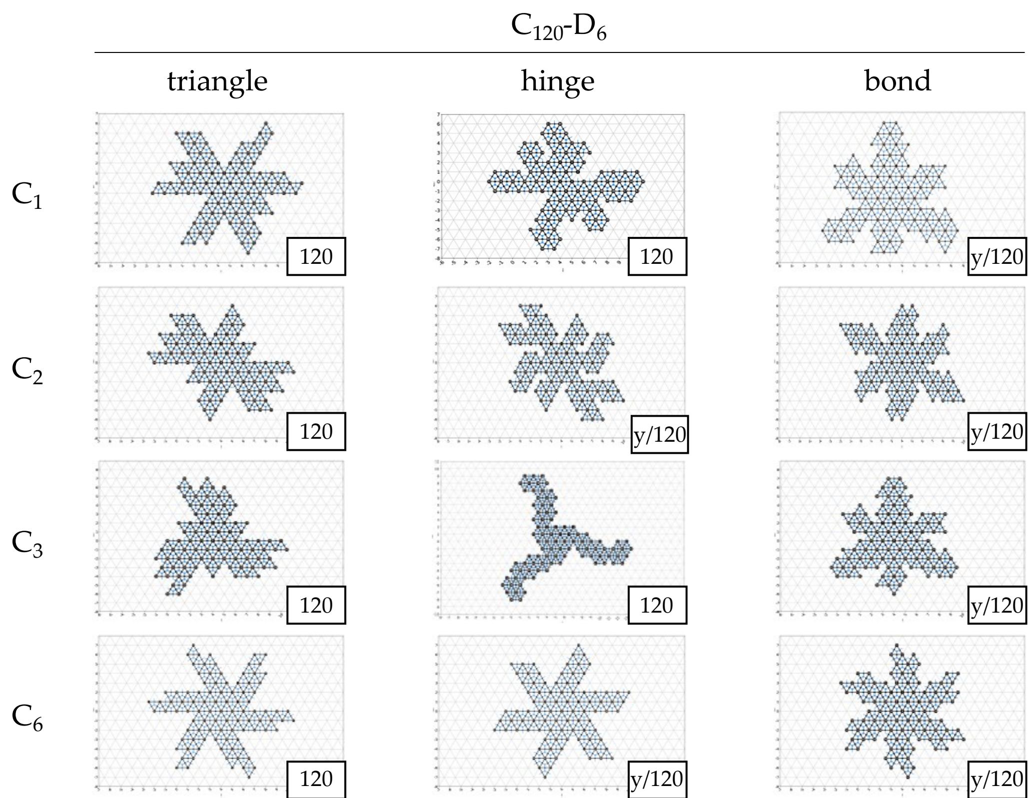


Figure 5.4: Collection of C_n unfoldings generated in the discovery run for C_{120}-D_6 , marked with a 'y' for hybrid unfoldings and the number of triangles placed.

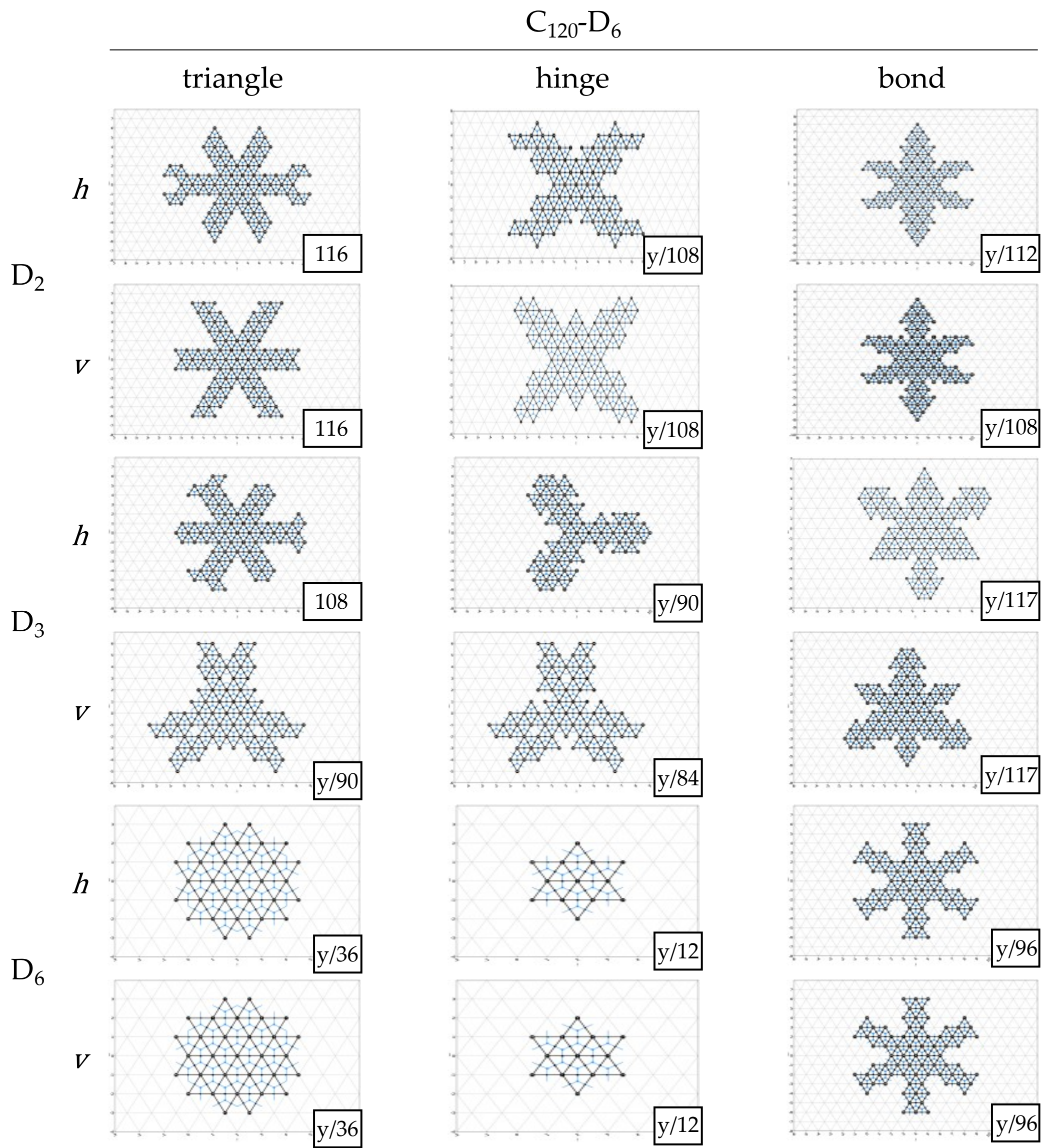


Figure 5.5: Collection of D_n unfoldings generated in the discovery run for C_{120}-D_6 , marked with a 'y' for hybrid unfoldings and the number of triangles placed.

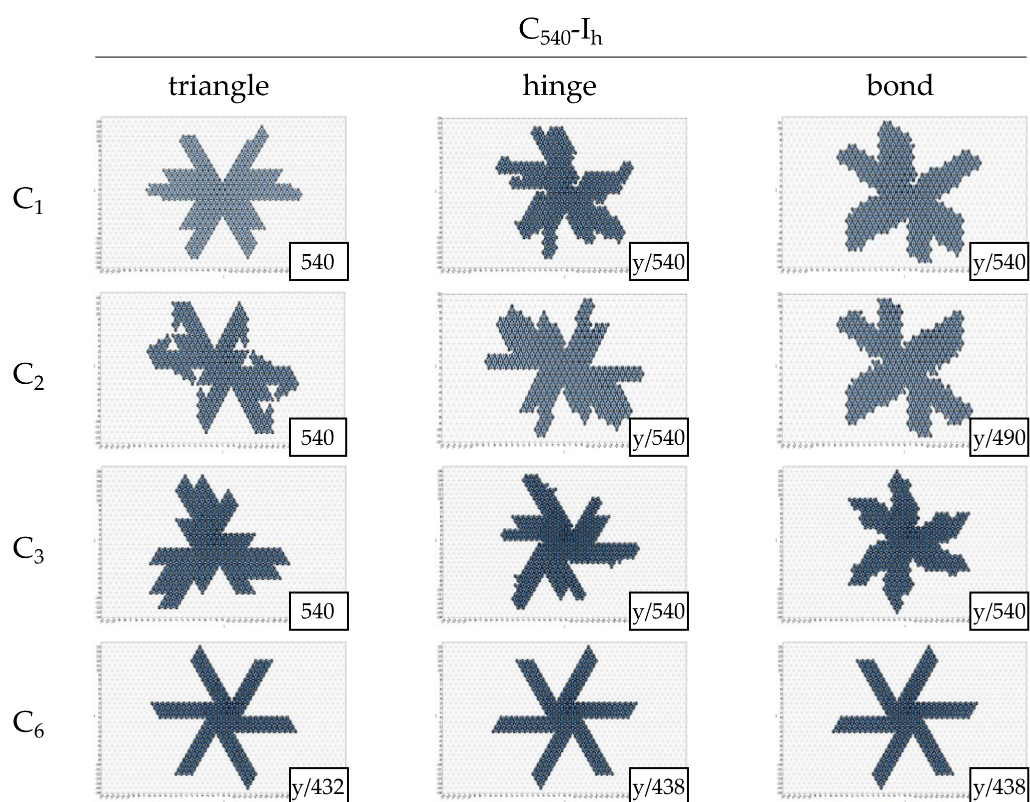


Figure 5.6: Collection of C_n unfoldings generated in the discovery run for C_{540}-I_h , marked with a 'y' for hybrid unfoldings and the number of triangles placed.

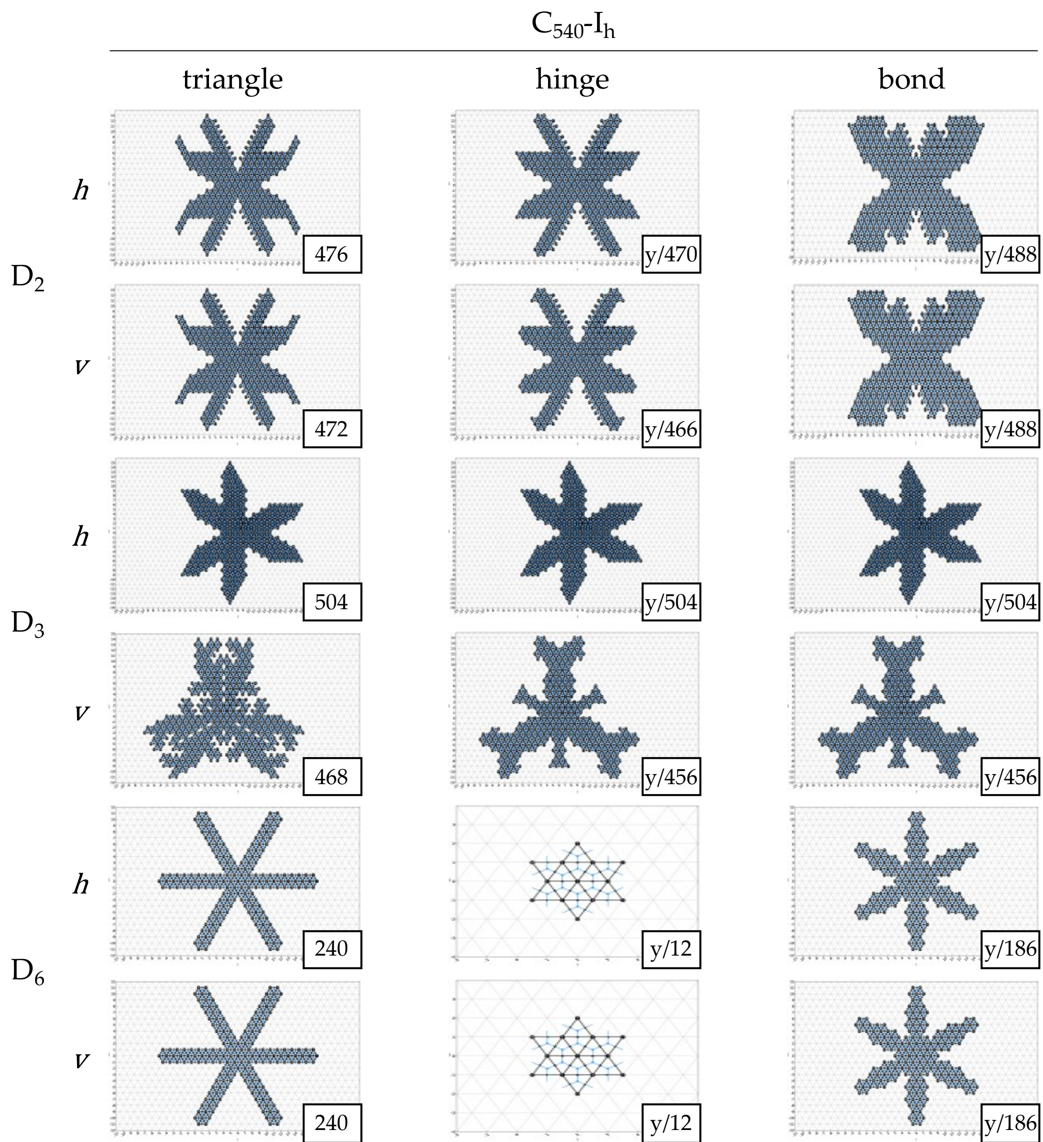
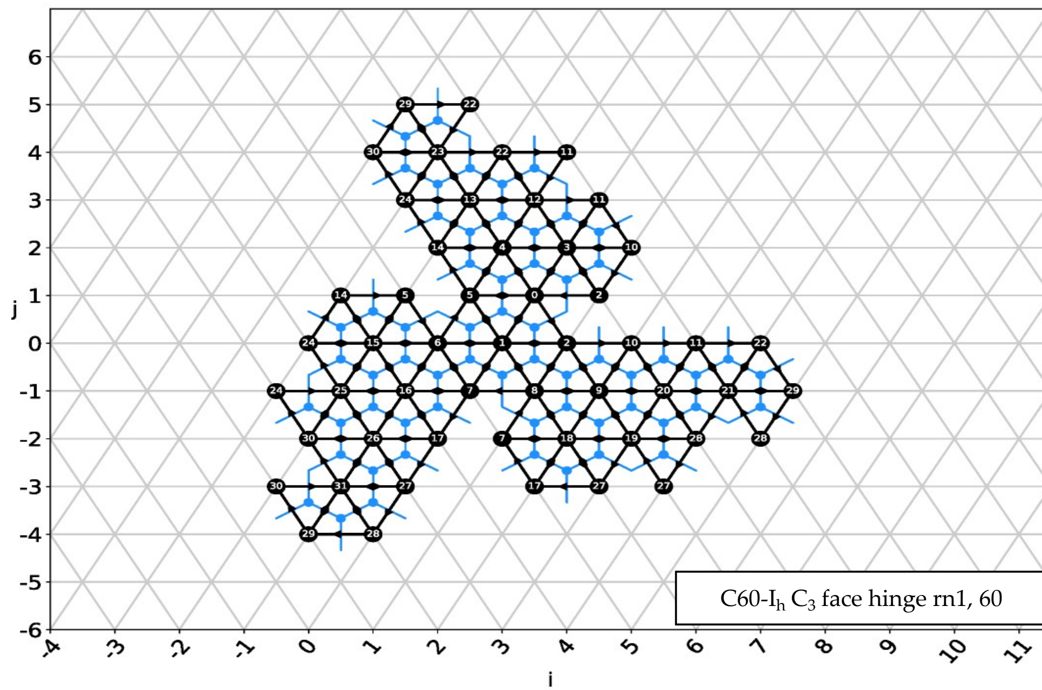


Figure 5.7: Collection of C_n unfoldings generated in the discovery run for C_{540}-I_h , marked with a 'y' for hybrid unfoldings and the number of triangles placed.



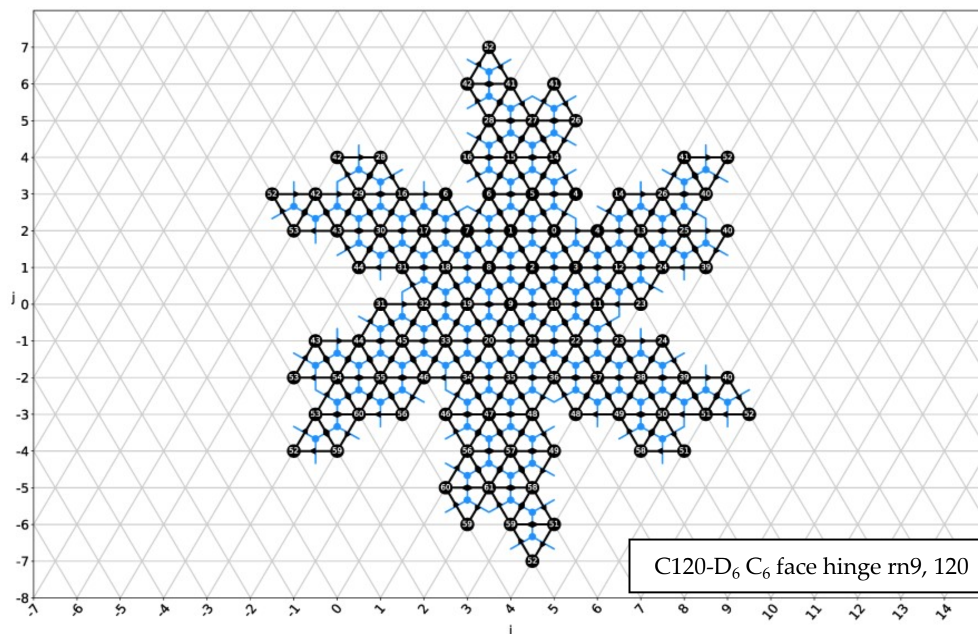


Figure 5.8: A showcase of valid generated unfoldings in the discovery run.

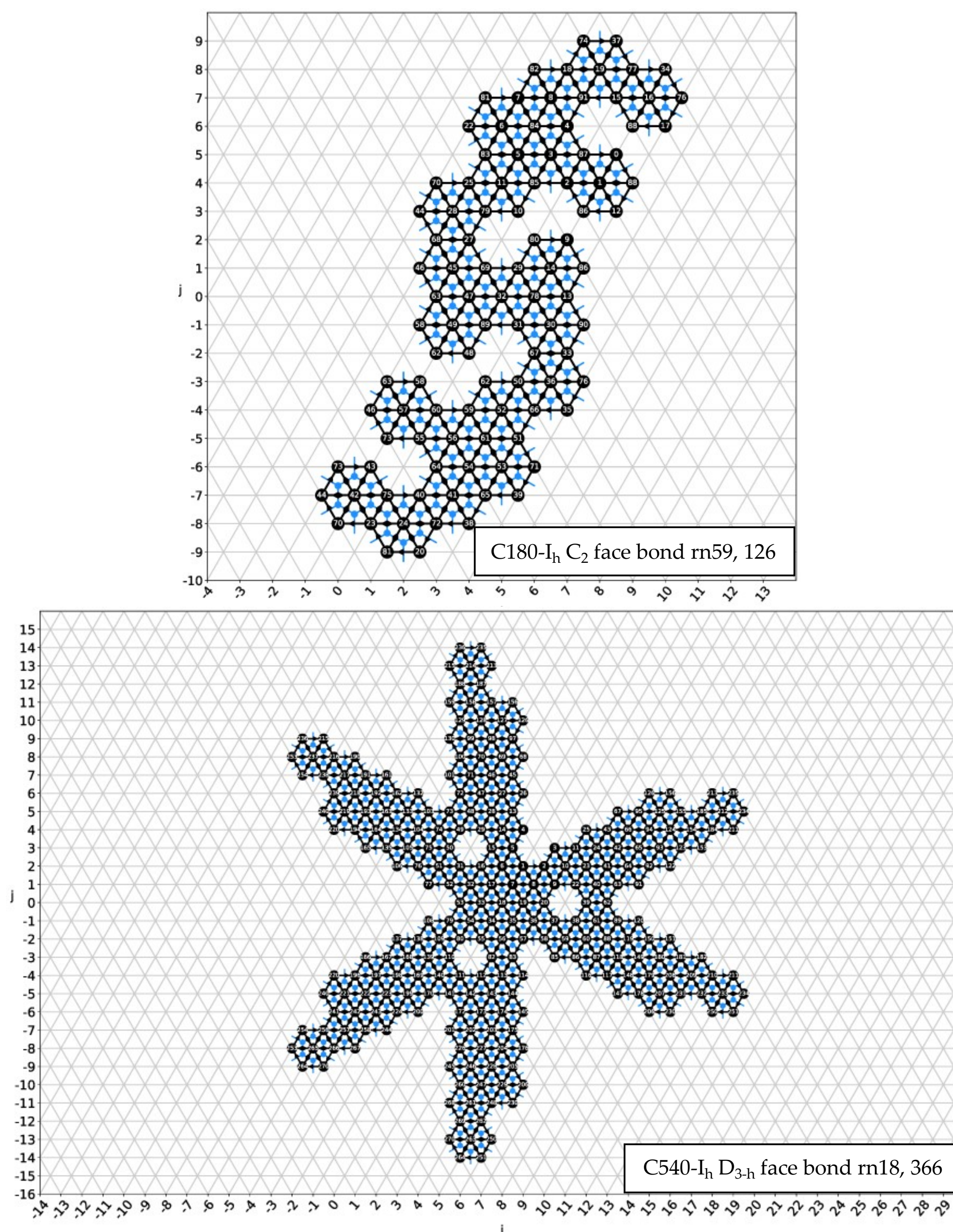


Figure 5.9: A showcase of invalid generated unfoldings in the discovery run.

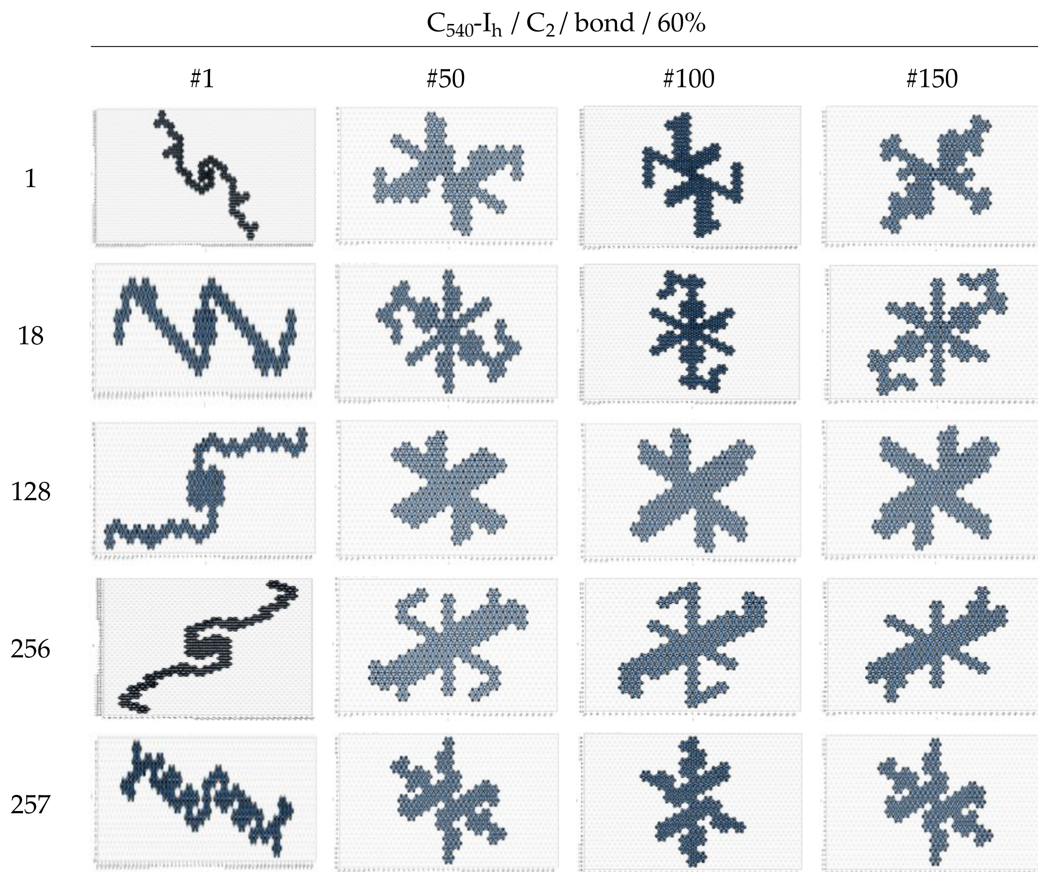


Figure 5.10: Collection of C_2, hinge-connected face unfoldings generated in the variety run for C_{540}-I_h . A distinct unfolding pattern appears to emerge for each of the root-nodes.

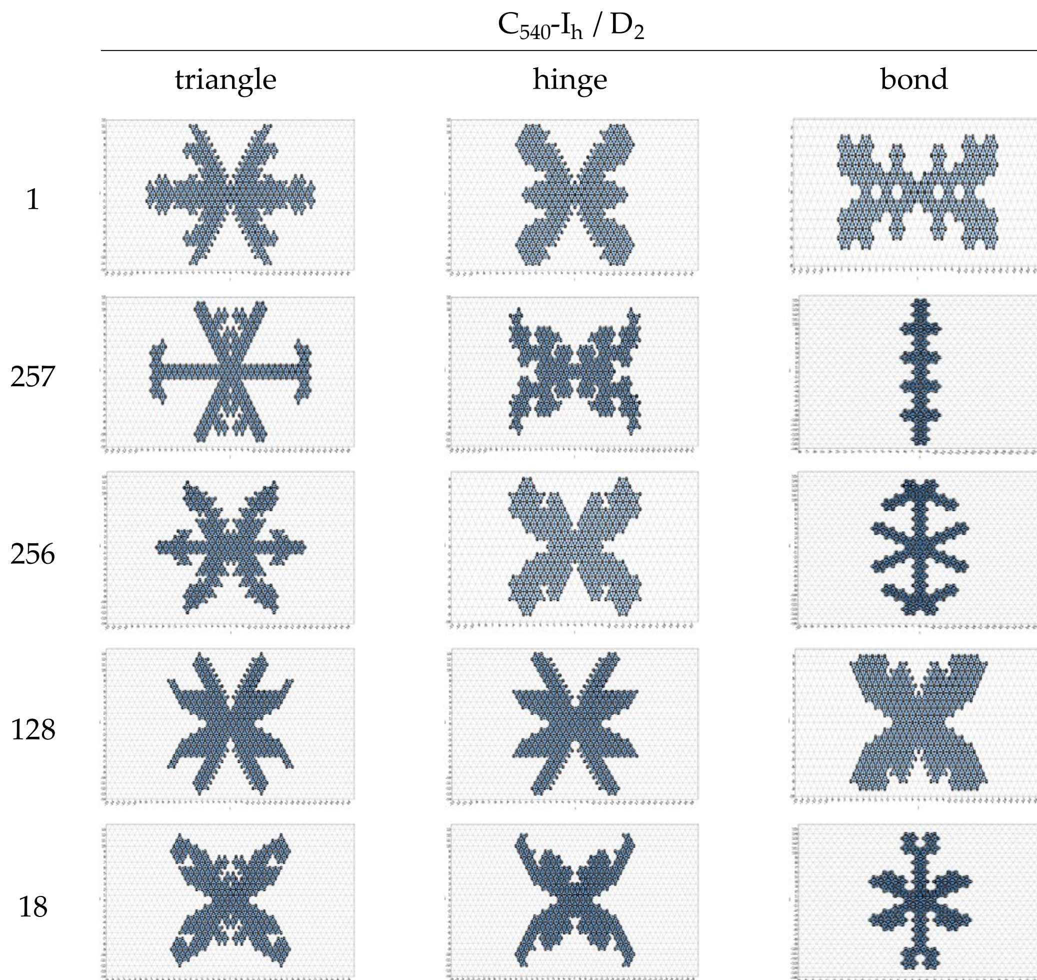


Figure 5.11: Collection of D_2 unfoldings generated in the variety run for C_{540}-I_h with the different placement objects.

5.1.2 *Performance*

Each run of the algorithm was performed on a single core of an Intel i7-8650U CPU which is rated at an internal clock speed 1.90-2.11GHz. To be able to evaluate the performance of the algorithm, the time taken and number of recursion (placement) steps were recorded for each parameter combination of the discovery run. Using these values, Tables 5.4 and 5.5 were created. The first lists the time it took for the algorithm to generate a valid C_3 hinge-connected face unfolding, since this is the symmetric category for which a valid unfolding was found for the highest number of isomers. In the second table the average recursion steps per second was calculated for each of the point groups across all isomers.

Isomer	C_{60}-I_h	C_{120}-D_6	C_{120}-D_{6d}	C_{120}-T_d	C_{140}-I_h	C_{180}-I_h	C_{540}-I_h
Time taken (s)	0.088	0.223	0.251	0.288	-	-	4.029
Recursion steps	7	32	11	11	38133	46559	711

Table 5.4: List of the time taken for each of the isomers to find its first valid C_3 hinge-connected unfolding in seconds. For C_{140}-I_h and C_{180}-I_h , no unfolding was found within the time limit.

Point group	C_1	C_2	C_3	C_6	D_2	D_3	D_6
Steps/s	450	55	51	51	34	29	11

Table 5.5: Average values for the recursion steps per second across all isomers for the different point groups.

From the first performance data in Table 5.4, it can be seen that for the small isomers a valid unfolding is found within milliseconds and very few recursion steps, while for the larger C_{540}-I_h isomer the time taken increases twenty-fold to around four seconds. Since the number of atoms does not increase twenty times as well, it can be concluded that for this combination of parameters, the time and steps it takes to find a valid unfolding does not increase linearly with the number of atoms. This may be due to the fact that the search space of unfoldings does not grow linearly with the number of atoms either, but more research would be required on a much wider range of different sized fullerenes before any definitive claims can be made.

Continuing to Table 5.5, it can be seen that the number of steps per second declines as the complexity of the point group increases. This is expected since the complexity of the code also increases, and more symmetry arcs have to be evaluated when n in C_n and D_n increases. When furthermore considering the data from Tables 5.2 and 5.3, it can be noted that the results for both C_n and D_n symmetry are consistent for the isomers up to C_{140}-I_h , at which point the number of atoms grows too large and the algorithm is unable to search enough of the recursion tree in the given time to find a valid unfolding. In other words, the limitations of the discovery run appear to be performance related and not to any incorrect functioning of the algorithm.

5.2 DISCUSSION

5.2.1 Unfoldings

In the discovery run, the algorithm was able to generate valid unfoldings using nothing but the fullerene bond-graph for at least one parameter combination for each isomer. The validity of the unfoldings can be confirmed upon closer inspection of the figures. Starting with Figure 5.8, for each arc on the periphery, there is a reversed counterpart with which it will align during autoassembly. Moreover, there are no disconnected parts in the unfoldings, as all atoms are bound. The twelve pentagons are also present, with the doubled nodes of the wedge cutouts as expected. Another thing to note is that the C_n symmetries are correct as well, with each of the molecular arms oriented in the same manner. For the C_{60}-I_h Buckeyball unfolding, he result closely resembles the precursor molecule that was synthesised by Scott et al. (2002), with three equivalent arms of atoms attached to the central hexagon via three pentagons. It is highly likely an exact match to the synthesised precursor-molecule exists in the generated unfoldings, although such a statement could as of yet only be confirmed visually.

Moving on to the incomplete unfoldings of Figure 5.9, it can be seen in the figure that there are only bond-connected faces present as per the constraints. The unfoldings were taken from a single recursion path of the discovery run of the depicted

isomers to exemplify the validity of intermediate unfolding states. The intermediate unfoldings also adhere to the point groups that were set, with clear C_2 rotational symmetry for the C_{180}-I_h unfolding, and three reflection planes for the D_3 symmetric unfolding of C_{540}-I_h . An interesting feature of the latter is the presence of three distinct hexagon shaped holes, which were formed by six bond-connected faces creating 'bridges' of atoms between the arms. It is unclear at this time whether such an unfolding would fold up to a valid fullerene cage. For this reason, a quantum chemical study could be made to see if precursors such as these, where curvature is induced by closing the hexagonal holes, can be autoassembled. In case it is found that this is not the case, the algorithm could be modified to detect whenever holes are introduced, and cut off the recursion branch if this the case. This could be done by identifying when faces 'glue' together as they form the bridges that create the holes, and disallowing this type of collision as well, for example. Regardless, it is possible holes in the unfolding are a new way to initiate the curvature in unfoldings. Holes in the unfolding are not exclusive to the larger fullerenes either, nor are they always hexagon shaped, as can be seen in the unfoldings (C_{60}-I_h , C_3, bond) of Figure 5.2, (C_{120}-D_6 , D_{6v}, triangle) of Figure 5.5, and (C_{540}-I_h , D_2, bond, root-node 1) of Figure 5.11.

An interesting observation from comparing Table 5.3 with Table 5.2 is that triangle unfoldings were able to succeed in finding a valid unfolding with parameter combinations where hybrid unfoldings were not. An example of this is seen in Figure 5.6 between the unfoldings (C_2, triangle) and (C_2, bond) which placed 540 and 490 triangles respectively. Although it is possible the bond-connected unfolding would be completed given more time, since bond unfoldings are also shown to have placed more triangles than there exclusive triangle counterparts in Figure 5.7 for D_{2h} and D_{2v} for example. In fact, this discrepancy might be simply due to the fact that the symmetric triangle unfolding has yet to reach a better solution which is more in line with the shape of the more successful bond-unfolding.

With regards to symmetry, the algorithm was able to successfully create valid C_n symmetry unfoldings for each isomer, and near complete unfoldings for all D_n point

groups. As is shown in Figure 5.4, for C_{120}-D_6 a valid unfolding for each type of C_n symmetry was found. The unfoldings clearly differ in shape based upon the chosen placement object, but all appear to adhere to a structure of a central compact core, with arms of atoms for each rotation. It can moreover be seen that the C_3, bond unfolding formed two sets of three arms, but because of the twists in the outer parts of the arms does not reach C_6 symmetry. In the figure, unfoldings with triangles as placement objects form a distinct triangular core, whilst unfoldings with faces form diamond or hexagonal structures, even when the unfolding is hybridised. Although this may seem unsurprising, it visually indicates the face unfoldings were able to reach between 70%-90% completion before triangles were needed to finish it.

Of note is the case of D_6 symmetry, for which none valid unfoldings were found and the search space was exhausted for most isomers. Starting with C_{60}-I_h in Figure 5.3, it can be seen that each of the possible parameter combinations yields twelve triangles, six appended to the starting hexagon. The reason for this is that the hexagon is surrounded by six hinge-connected pentagons. This eliminates further possibilities for D_6 for all cases since the next placement step would block the wedge cut-outs of the pentagons. The only alternative is to break D_6 symmetry and continue with C_6 symmetry instead. This can be confirmed by examining the results for C_6 in Figure 5.2 and seeing how the six arms are separated by the six cut-outs. It is notable that the C_6 face unfoldings get stuck at the same 12 triangles as for D_6, but the hybridisation of the unfolding is able to create partial arms until the options run out at 48 triangles. A similar occurrence can be seen for D_6 symmetry of C_{120}-D_6 in Figure 5.5, where progress is halted at 36 triangles as it reaches a ring of pentagons that surround the core. The fact that these unfoldings were not found does not necessarily indicate that the unfoldings do not exist, but it does indicate that they are impossible to create with faces as symmetry sites. Therefore, future work should try to implement other site-symmetries as well, being triangles (atoms) and edges (arcs), to evaluate the effect upon the possible unfolding paths for the various planar point groups C_n and D_n.

One way to progress the unfoldings that appear unable to 'break out' of the core structure would be to create flexibility in the placement constraints similar to the placement priorities that were created to have flexibility in the chosen placement-object depending on the state of the unfolding. In this way, a list of point group priorities could also be created, which would allow the algorithm to break out of a dead-end using the Cn equivalent of the given D_n symmetry. An additional step would be to be able to switch back to maximal symmetry again after valid placement steps are made, to try and create D_n unfoldings that will in the worst case end up as C_n instead.

Differing combinations of parameters appear to be an effective way to direct the shape of the unfolding. This is exemplified well for the bigger C_{540}-I_h isomer shown in Figures 5.6 and 5.7. The higher n is in terms of symmetry, the more strict and clear cut the periphery has to become. Although this rule appears to be broken by the C_3 hybrid bond unfolding, a closer look reveals there are in fact two sets of three symmetric arms instead of six identical ones. There also appears to be a greater level of homogenisation of the final result as the level of symmetry increases. This could be simply due to the fact that there are less possibilities for placement at each recursion level, thereby narrowing the possible results. However, given the relatively small recursion space the algorithm has searched, it can also be a byproduct of the fact that the solutions lie close to each other in terms of recursion steps, meaning it is possible they share a significant portion of intermediate states (parent and ancestor nodes) before diverging.

When it comes to variety Figure 5.10 however, Figure 5.11 make it clear that the algorithm does traverse a wide variety of paths that that lead to unique (intermediate) unfoldings. An interesting observation that can be made from Figure 5.10 is that the unfoldings generated for the same root-node appear to shape themselves along a pattern. In its current implementation, the algorithm seems to start by extending outward as far as possible from the central core, while clustering atoms along the arms whenever possible as well. The results range from the 'orange peel' type unfoldings as in column #1, to the bar like structures of rows 128 and 256, as well

as interesting two-armed shapes with only a single central hexagon as per rows 1 and 257 from $\#50$ and up.

Another point of note is that some of the unfoldings appear to be rotations or reflections of one another. It is almost certain then that the algorithm in its current form generates stereoisomeric precursor-molecules: which means they have identical molecular formulas as well as arrangements of atoms. They differ from each other only in the spatial orientation of groups in the molecule. In this way, $(18, \#100)$ and $(18, \#150)$ of Figure 5.10 appear to be stereoisomers, for example. Since is there is no functional difference between stereoisomers, future work on the algorithm could exclude them, either by filtering them out after the generation of unfoldings is complete, or by identifying them in intermediate recursion steps and cutting the branch if the stereoisomer already exists in the list of outputs. This could be done as follows: when an object is placed, after its recursion tree is exhausted and the algorithm unwinds back to the node where the object was placed, the object work-arc is removed from the workset, thereby making it unavailable for the other recursion paths to place the object of the work-arc in the same location again in this particular configuration. In theory, that should remove most of the numberous stereoisomeric duplicate unfoldings the algorithm generates.

Between placement objects, the pattern of unfolding silhouettes that emerged for the root-nodes is less apparent, as can be seen in Figure 5.11. What can be stated is that (D_2) hinge-connected face unfoldings are more similar to triangle unfoldings than bond-connected face unfoldings are to either. This makes sense, as the faces with hinge connections for D_2 grow outward with a maximum of $N_{faces} \times 4$ triangles for hexagons (3 for pentagons) due to the overlapping hinge triangles, while this number is $N_{faces} \times 6$ for bond-connected hexagons (5 for pentagons). As such, the more compact structure of hinge-connected face unfoldings appears to be in line with triangle unfoldings more closely, while bond-connections more elongated structures, as was also the case in Figure 5.10.

In terms of the predicted autoassembly success of the generated unfoldings, it can be seen from the figures that the unfoldings that are the most complete are compact, and

will attempt to tile the Eisenstein plane with triangles until either the pentagon cut-outs or symmetry constraints force space between different parts of the unfolding. The resulting molecular arms mimic those of the synthesised precursor by Scott et al. (2002), and promise the possibility of creating parts of the unfolding in parallel. The second indicator of autoassembly success was the minimal adjacency rule for pentagons, which states that the most stable chemical geometry is achieved when pentagons do not share a bond with any other pentagon. Considering the collections of unfoldings shown, it is unclear what the precise effects of the parameters are on this. However, what can be said is that if the core of the unfolding is large enough as to separate the pentagons with at least one hexagon, and subsequently creating as many arms as possible from each of the points where the pentagons are, this rule is adhered to the most. From this, it can be speculated that it might be preferable to create as many arms as possible given the size of the core of the unfolding, thereby placing the maximum possible number of pentagons separated by hexagons for the unfolding.

5.2.2 *Performance*

There are two points to address when it comes to explaining the speed and performance of the algorithm. The first is the programming language, as Python is considered slow in general, as well as notoriously bad at recursion. The reason for this is that Python has an overhead on each function call, which the recursion used in the algorithm can potentially generate tens if not hundreds at a time of. The reason for this is that Python is a dynamically typed language, which means the types variables have (e.g. integer, string) are checked by the interpreter at runtime, and are moreover allowed to change over the course of the runtime, which causes additional runtime latency. In general then, the performance of the recursive algorithm in its Python implementation is slow, with a total average of 41 steps per second across all isomers and point group combinations.

Notwithstanding, there are also parts of the algorithm itself that result in sub-optimal performance. The main source of redundancy lies in the constraints, specifically

the symmetry constraints used in `sym_is_placeable()`. It is possible that the purpose of functions such as `is_interim_placed()`, `filter_sym_arcs()` as well as the hinge dependent arc removal done by `face_is_placeable()` can be combined in ways so as to avoid having to go through the list of symmetry arcs multiple times. It could be interesting to re-think the required constraints for the arcs, and see if any overlap exists in the current way edge-cases are handled by the constraint functions.

What is more, in the ideal situation the symmetry of placement objects is not determined by means of their Eisenstein coordinates, but instead only by their abstract symmetry information. This would require more extensive knowledge on the calculation of abstract groups however, and was as such outside of the scope of this project. A starting point could be a more detailed study of the spiral algorithm by Wirz2018, to see how the graph information could be used during the unfolding process to identify symmetry equivalent objects efficiently. There are also separate checks in all of the major functions to identify the placement object and point group of the run. This could be integrated better by perhaps creating an unfolding class that shares the variables of the run as to avoid any unnecessary checks. On the other hand, the current implementation of the code was made to be as clear as possible, and to that end does serve as a useful template for future work.

5.3 FUTURE WORK

5.3.1 Unfoldings

In terms of increasing the chances of autoassembly success, the first step for future work is to make use of the study done by Heuser et al. (2021) that explored the fold up method for a C_{60}-I_h precursor-molecule using quantum chemical simulations. Unfoldings of varying shapes and parameter combinations generated by the algorithm presented here should be used as input for the autoassembly simulations. This would allow for the validation of the unfoldings in terms of autoassembly, as well as possibly uncover shape characteristics that contribute to the chance of

success of autoassembling the precursor-molecule. Any discoveries made could subsequently be implemented in the `is_valid_unfolding()` function presented here, to provide more control on the recursion paths and thereby unfoldings that are produced.

The collaboration would start a feedback loop of evaluating the autoassembly of generated unfoldings, identifying point of improvement and desirable characteristics of the unfolding, implementing the improvements, and trying the unfoldings for autoassembly again. A possible result is that unfoldings are preferred to have only completely equal molecular arms for example, instead of two sets of different arms as was the case for many of the $n = 2, 3$ symmetric unfoldings such as those in Figure 5.5. The algorithm should therefore allow for more control of the final shape of the generated unfoldings, possible by computing shape variables of the polygon such as interior angles or the second moment of area to assess the width or elongation of the unfolding, for example. With even more work, the discovery of the optimal locations for placement of the halogens (i.e. the 'glue' atoms) in the precursor-unfoldings should be implemented as well, possibly on the basis of the shape characteristics of valid unfoldings that were mentioned before.

Practically, when generating unfoldings the shape characteristics that are used as generation constraints can be influenced by choosing between (sets of) valid work-arcs at each placement step, and instead of always placing a valid set of arcs, only place arcs that are in line with the desired shape characteristics. To this end, work-arcs would need to be assigned properties that inform the algorithm of the effect the corresponding placement object has on the shape. For example, a work-arc may place a face in a specific Eisenstein direction that elongates the profile of the unfolding, while another choice may create a more compact precursor-molecule instead on the basis of its calculated moment or distance to the centre of the unfolding and any surrounding placement objects.

Regarding recursion, the generation constraints should impact which unfoldings are collected and which are discarded. Hence, the current validity check in the recursion algorithm should be expanded to be used as a tool to define and implement

generation constraints that can cull the large search space of possible precursor-unfoldings. Put differently, apart from control on the micro level of single placement objects, another level of control comes by deciding which recursion branches to cut in the tree and which to traverse down further. If a particular direction in the tree leads to unwanted shape characteristics no further effort should be put into the possible unfoldings that form at that branch.

5.3.2 *Performance*

In terms of performance, the first step of action should be to do a one-to-one translation of the algorithm into C++ , which should yield an improvement on the performance between $2\times$ and $400\times$. The step after that would be to look at eliminating the many new memory allocations and copies made of recursion states and state variables, by changing the state in place before recursing downwards and then 'repairing' upon return to the parent node by undoing the placement step, before continuing to the next work-arc and creating additional child nodes in the recursion tree, combined with the C++ implementation this could increase the speedup factor to $1000\times$. After that is done, the next step would be to detect dead ends in the recursion tree earlier and cut off branches without solutions as high up as possible, thereby culling the search space. The last point of improvement would be to parallelise the code to make better use of the available number of cores in the hardware that is used, but should be the last step to implement.

6 CONCLUSION

In this thesis, a recursive algorithm was created with the objective of generating all possible planar fullerene precursor-molecules using nothing but intrinsic geometry in the form of the dual bond-graph for any fullerene isomer given enough time. The algorithm was run on a single core of an Intel Core i7 CPU with a clockspeed of 1.9-2.11GHz for 30 minutes or until the search space was exhausted for seven distinct fullerene isomers of varying sizes and symmetries. This resulted in a total of 700 parameter combinations including root-nodes that were run to discover valid unfoldings. A second run with a time limit of 10 minutes was done which introduced placement priorities to create hybrid unfoldings by starting with the placement of faces until a dead end was reached for unfoldings that were at least 65% complete, at which point the algorithm switched to triangles to try and close out the unfoldings. A third run was done by generating 150 unfoldings with a threshold of 60% for each possible root-node of the C_{540}-I_h isomer to explore the variety of unfolding paths and confirm the algorithm recurses through the full search space of precursor-molecules given enough runtime.

With the combined results of the first and second run, the algorithm was able to successfully create valid precursor-unfoldings of the isomers. Specifically, valid unfoldings were found for non-symmetric triangle unfoldings, as well as hinge-connected and bond-connected face unfoldings. Moreover, valid symmetric unfoldings were found for at least one of the isomers given all planar C_n and D_n point groups with $n = 2, 3, 6$, given that they existed in the search space of the parameter combination. For the selected isomers with more than 120 atoms, the algorithm was unable to find valid unfoldings for all possible C_n symmetries due to the time limit being reached. What is more, no valid D_n symmetry unfoldings were found as either

the search space was exhausted due to the pentagon configuration of the isomer blocking further placement steps.

From the collections of unfoldings generated in the third run, it could be visually confirmed that the algorithm traverses a wide range of unfolding paths while searching for valid unfoldings. Moreover, unfolding shapes were found to vary based upon the chosen placement object and root-node as expected. In terms of the ability of the algorithm to produce all possible unfoldings, there are no signs that the algorithm would not be able to recurse through all possible paths and produce all possible unfoldings for any N-atomic fullerene isomer if they exist, given enough runtime.

Returning now to the problem statement:

P.1 Generates unfoldings

  (a) Generates a carbon fullerene precursor molecule unfolding from nothing but the fullerene bond-graph.

  (b) Generates all possible unfoldings for any N-atom fullerene isomer given enough computation time.

P.2 Encodes generation constraints

  (a) Generates unfoldings that are comprised of full cubic faces, meaning hexagons and pentagons.

  (b) Generates unfoldings with (maximal) planar symmetry.

It can be concluded that all items of the problem statement were successfully reached, although there is a lot of room for improvement regarding P.1b. The limiting factor for the current implementation of the algorithm is its performance, with an average of 41 recursion steps per second taken across all isomers and all point group combinations. As a result, out of the 119 combinations only 10 non-hybrid and 24 hybrid valid solutions were found, whereas the time limit was reached in 45 cases, and for the remaining 40 cases the search space was exhausted. Notwithstanding,

upon investigation of the intermediate results it was confirmed that the partial unfoldings were valid. For this reason it can be stipulated that the algorithm would be able to find valid unfoldings given faster hardware and more runtime. The largest boost in performance can be achieved by implementing the algorithm in C++ instead of Python, which should yield an increase in speed of a factor between 2 and 400, or possibly 1000 with additional optimisations of the algorithm itself. Notwithstanding, the results presented in this thesis are in line with the objective of the project, which aimed to create a correct, understandable, and functional algorithm which produces valid precursor-unfoldings under a variety of generation constraint parameters.

Future work on the algorithm should start by addressing the performance issues with a direct translation of the algorithm into C++ to be able to generate more valid unfoldings for more isomers, in much less time. Moreover, optimisations in the algorithm can be made by repairing recursion states upon unwinding instead of copying them to memory, as well as streamline the placement constraints by identifying commonalities between the edge-cases that the constraints solve for. Further work includes the determination of symmetric placement object on the basis of their abstract group information, as well as culling the search space by identifying stereoisomeric unfoldings and implementing new validity constraints on the basis of possible shape characteristics calculated in intermediate recursion states. Lastly, a collaboration with the work from Heuser et al. (2021) should be started to discover new rules of thumb that can increase the autoassembly success of the generated unfoldings, and implement any constraints found this way.

BIBLIOGRAPHY

1. Albertazzi, E., Domene, C., Fowler, P. W., Heine, T., Seifert, G., Alsenoy, C. V. & Zerbetto, F. (1999). Pentagon adjacency as a determinant of fullerene stability. *Physical Chemistry Chemical Physics*, *1*(12), 2913–2918. https://doi.org/10.1039/a901600g

2. Avery, J. (2020). *Folding carbon: A calculus for molecular origami* [Seminar slides].

3. Avery, J. (2021). *Carbon manifolds: Computing precursors and recipes for rational synthesis of fullerenes*. Retrieved January 2, 2021, from https://www.nbi.dk/~avery/CARMA/Task-R/

4. Bakry, R., Vallant, R. M., Najam-ul-Haq, M., Rainer, M., Szabo, Z., Huck, C. W. & Bonn, G. K. (2007). Medicinal applications of fullerenes. *International Journal of Nanomedicine*, *2*(4), 639–649. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2676811/

5. Becker, L., Poreda, R. J. & Bunch, T. E. (2000). Fullerenes: An extraterrestrial carbon carrier phase for noble gases. *Proceedings of the National Academy of Sciences*, *97*(7), 2979–2983. https://doi.org/10.1073/pnas.97.7.2979

6. Brinkmann, G., Fowler, P. & Yoshida, M. (1998). New non-spiral fullerenes from old: Generalised truncations of isolated pentagon-triple carbon cages. *MATCH Commun Math Comput Chem*, (38), 7–17. https://match.pmf.kg.ac.rs/electronic_versions/Match38/match38_7-17.pdf

7. Brinkmann, G., Goedgebeur, J. & McKay, B. D. (2012). The generation of fullerenes. *Journal of Chemical Information and Modeling*, *52*(11), 2910–2918. https://doi.org/10.1021/ci3003107

8. Buseck, P. R., Tsipursky, S. J. & Hettich, R. (1992). Fullerenes from the geological environment. *Science*, *257*(5067), 215–217. https://doi.org/10.1126/science.257.5067.215

9. Buseck, P. R. (2002). Geological fullerenes: Review and analysis. *Earth and Planetary Science Letters*, *203*(3-4), 781–792. https://doi.org/10.1016/s0012-821x(02)00819-1

10. Chae, S.-R., Hotze, E. M. & Wiesner, M. R. (2014). Possible applications of fullerene nanomaterials in water treatment and reuse. *Nanotechnology applications for clean water* (pp. 329–338). Elsevier. https://doi.org/10.1016/b978-1-4557-3116-9.00021-4

11. Deza, M., Sikiric, M. & Fowler, P. (2009). The symmetries of cubic polyhedral graphs with face size no larger than 6. *MATCH*.

12. Fischer, J. E., Heiney, P. . & Amos B. Smith, I. (1992). Solid-state chemistry of fullerene-based materials. *Acc. Chem. Res.*, *25*, 112–118. https://doi.org/10.1021/ar00015a003

13. Fowler, P. W. (2006). *An atlas of fullerenes*. Dover Publications.

14. Goldberg, M. (1937). A class of multi-symmetric polyhedra. *Tohoku Mathematical Journal, First Series*, *43*, 104–108.

15. Hasheminezhad, M., Fleischner, H. & McKay, B. D. (2008). A universal set of growth operations for fullerenes. *Chemical Physics Letters*, *464*(1-3), 118–121. https://doi.org/10.1016/j.cplett.2008.09.005

16. Heuser, B. (2020). *Folding carbon: Computational study of the auto assembly of existing fullerene precursor molecules and building towards the fast automated quality assessment of an arbitrary fullerene structure* (Master's thesis). University of Copenhagen. Blegdamsvej 17.

17. Heuser, B., Mikkelsen, K. V. & Avery, J. E. (2021). Simulating fullerene polyhedral formation from planar precursors. *Phys. Chem. Chem. Phys.*, *23*, 6561–6573. https://doi.org/10.1039/D0CP04901H

18. Howard, J. B., McKinnon, J. T., Makarovsky, Y., Lafleur, A. L. & Johnson, M. E. (1991). Fullerenes c60 and c70 in flames. *Nature*, *352*(6331), 139–141. https://doi.org/10.1038/352139a0

19. Jensen, F. (2017). *Introduction to computational chemistry, 3rd edition*. John Wiley & Sons, Ltd.

20. Kabdulov, M. A., Amsharov, K. Y. & Jansen, M. (2010). A step toward direct fullerene synthesis: C60 fullerene precursors with fluorine in key positions. *Tetrahedron*, *66*(45), 8587–8593. https://doi.org/10.1016/j.tet.2010.09.055

21. Krätschmer, W., Lamb, L. D., Fostiropoulos, K. & Huffman, D. R. (1990). Solid c60: A new form of carbon. *Nature*, *347*(6291), 354–358. https://doi.org/10.1038/347354a0

22. Kroto, H. W., Heath, J. R., O'Brien, S. C., Curl, R. F. & Smalley, R. E. (1985). C60: Buckminsterfullerene. *Nature*, *318*(6042), 162–163. https://doi.org/10.1038/318162a0

23. Langa, F. & Nierengarten, J.-F. (2007). *Fullerenes: Principles and applications*. RSC Publishing.

24. Majewski, M. A. & Stępień, M. (2018). Bowls, hoops, and saddles: Synthetic approaches to curved aromatic molecules. *Angewandte Chemie International Edition*, *58*(1), 86–116. https://doi.org/10.1002/anie.201807004

25. Mani, P. (1971). Automorphismen von polyedrischen graphen. *Mathematische Annalen*, *192*(4), 279–303. https://doi.org/10.1007/bf02075357

26. Marchesan, S., Ros, T. D., Spalluto, G., Balzarini, J. & Prato, M. (2005). Anti-HIV properties of cationic fullerene derivatives. *Bioorganic & Medicinal Chemistry Letters*, *15*(15), 3615–3618. https://doi.org/10.1016/j.bmcl.2005.05.069

27. Mojica, M., Alonso, J. A. & Méndez, F. (2013). Synthesis of fullerenes. *Journal of Physical Organic Chemistry*, *26*(7), 526–539. https://doi.org/10.1002/poc.3121

28. Nimibofa, A., Newton, E. A., Cyprain, A. Y. & Donbebe, W. (2018). Fullerenes: Synthesis and applications. *Journal of Materials Sciecne Research*, *7*(3), 22–36. https://doi.org/doi:10.5539/jmsr.v7n3p22

29. Õsawa, E. (1970). Superaromaticity. *Kagaku (Chem)*, (25), 854–863.

30. Otero, G., Biddau, G., Sánchez-Sánchez, C., Caillard, R., López, M. F., Rogero, C., Palomares, F. J., Cabello, N., Basanta, M. A., Ortega, J., Méndez, J., Echavarren, A. M., Pérez, R., Gómez-Lor, B. & Martín-Gago, J. A. (2008). Fullerenes from aromatic precursors by surface-catalysed cyclodehydrogenation. *Nature*, *454*(7206), 865–868. https://doi.org/10.1038/nature07193

31. Schwerdtfeger, P., Wirz, L. & Avery, J. (2013). Program fullerene: A software package for constructing and analyzing structures of regular fullerenes. *Journal of Computational Chemistry*, *34*(17), 1508–1526. https://doi.org/10.1002/jcc.23278

32. Schwerdtfeger, P., Wirz, L. N. & Avery, J. (2015). The topology of fullerenes. *WIREs Computational Molecular Science*, *5*, 96–145. WIREs Comput Mol Sci 2015, 5:96–145. doi: 10.1002/wcms.1207. https://doi.org/10.1002/wcms.1207

33. Scott, L. T. (2004). Methods for the chemical synthesis of fullerenes. *Angewandte Chemie International Edition*, *43*(38), 4994–5007. https://doi.org/10.1002/anie.200400661

34. Scott, L. T., Boorum, M. M., McMahon, B. J., Hagen, S., Mack, J., Blank, J., Wegner, H. & de Meijere, A. (2002). A rational chemical synthesis of C_{60}. *Science*, *295*, 1500–1502.

35. Stankevich, I. V., Nikerov, M. V. & Bochvar, D. A. (1984). The structural chemistry of crystalline carbon: Geometry, stability, and electronic spectrum. *Russian Chemical Reviews*, *53*(7), 640–655. https://doi.org/10.1070/rc1984v053n07abeh003084

36. Wentrup, C. (2014). Flash (vacuum) pyrolysis apparatus and methods. *Australian Journal of Chemistry*, *67*(9), 1159–1165. https://doi.org/https://doi.org/10.1071/CH14096

37. Wentrup, C. (2017). Flash vacuum pyrolysis: Techniques and reactions. *Angewandte Chemie International Edition*, *56*(47), 14808–14835. https://doi.org/10.1002/anie.201705118

38. Whitney, H. (1932). Non-separable and planar graphs. *Transactions of the American Mathematical Society*, *34*(2), 339–339. https://doi.org/10.1090/s0002-9947-1932-1501641-2

39. Wirz, L. N. (2015). *Graph theoretic and electronic properties of fullerenes & biasing molecular modelling simulations with experimental residual dipolar couplings* (Doctoral dissertation). Massey University. Tennent Drive Palmerston North 4474 New Zealand.

40. Wirz, L. N., Schwerdtfeger, P. & Avery, J. E. (2018). Naming polyhedra by general face-spirals – theory and applications to fullerenes and other polyhedral molecules. *Fullerenes, Nanotubes and Carbon Nanostructures*, *26*(10), 607–630. https://doi.org/10.1080/1536383x.2017.1388231

41. Woods, P. (2020). The discovery of cosmic fullerenes. *Nature Astronomy*, *4*(4), 299–305. https://doi.org/10.1038/s41550-020-1076-5

A ADDITIONAL UNFOLDINGS

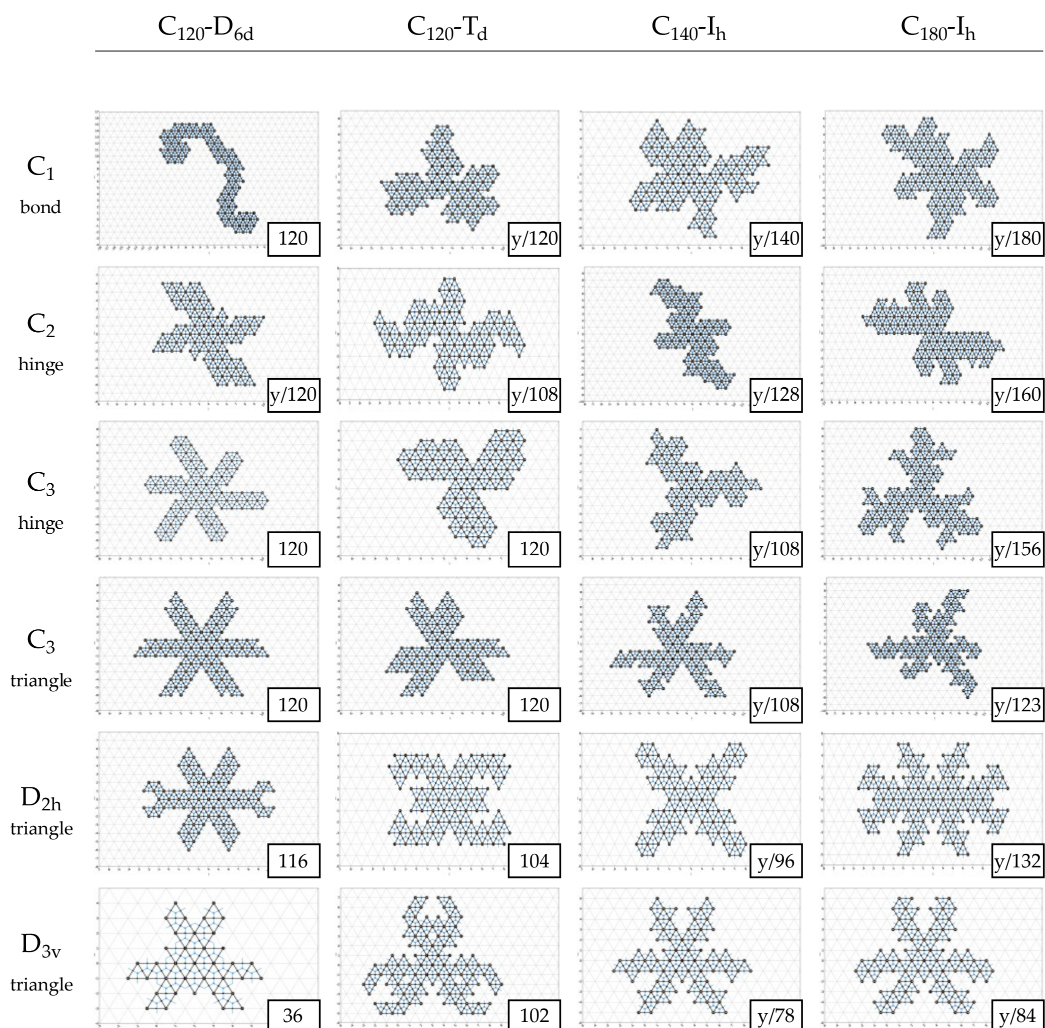


Figure A.1: A selection of unfoldings for the remaining isomers that were not discussed in the results section for completeness.