Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Abdulmelik Mohammed

Combinatorial Algorithms for the Design of Nanoscale Systems

Master's Thesis
Espoo, Jan 3, 2014

Supervisor: Professor Pekka Orponen
Advisor: Dr. Eugen Czeizler

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author: Abdulmelik Mohammed

Title:
Combinatorial Algorithms for the Design of Nanoscale Systems

Date: Jan 3, 2014 **Pages:** 72

Major: Information and Computer Science **Code:** T-79

Supervisor: Professor Pekka Orponen

Advisor: Dr. Eugen Czeizler

Over the past 30 years, DNA, with its exquisitely specific Watson-Crick base pairing rules, has found a novel use as a nanoscale construction material in *DNA nanotechnology*. *DNA origami* is a popular recent technique in DNA nanotechnology for the design and synthesis of DNA nanoscale shapes and patterns. DNA origami operates by the folding of a single long strand of DNA called a *scaffold* with the help of numerous shorter strands of DNA called *staples*. Recently, DNA origami design for polyhedral beam-frameworks has been proposed where a scaffold strand is conceptually routed over the beams of a polyhedron so that complementary strands potentially fold the scaffold to the framework in a solution.

In this work, we modelled the problem of finding a *scaffold routing path* for polyhedral frameworks in graph-theoretic terms whereby the routing path was found to coincide with a specific type of Eulerian trail, called an *A-trail*, on the polyhedral skeleton. We studied the complexity of deciding whether an A-trail exists with an emphasis on rigid triangular frameworks or equivalently on plane triangulations. While the decision problem was found to be NP-complete in general, we learned that Eulerian triangulations always have A-trails if a long standing conjecture by Barnette on the Hamiltonicity of bipartite cubic polyhedral graphs holds.

Given the general NP-completeness result, we developed a backtracking search algorithm for finding A-trails. To improve the backtrack search, we introduced an enumeration heuristic, tuned in particular to Eulerian triangulations, to schedule the nodes in the search tree. The algorithm, guided by the heuristic, efficiently found A-trails for a family of Eulerian triangulations as well as a family of braced grid graphs. Furthermore, we implemented a software package, BScOR (Beam Scaffolded-Origami Routing), which generates an A-trail, or equivalently a scaffold routing path, given a three-dimensional object description in a Polygon File Format.

Keywords: DNA nanotechnology, DNA origami, Polyhedra, Graphs, Combinatorial algorithms

Language: English

Acknowledgements

First and for most, I would like to express my deep gratitude to my supervisor Professor Pekka Orponen, and my advisor Dr. Eugen Czeizler, for giving me the opportunity to work on a challenging, yet engaging, project that I present in this thesis. I also wish to thank them both for their continuous support in all phases of the project and for their careful review of the thesis.

I further wish to extend my gratitude to Professor Björn Högberg for his collaboration, and for defining the problem in DNA nanotechnology that is the motivation of the work presented in this thesis. I also wish to thank my project co-worker, Gilberto Garcia Gape, for implementing parts of the software package that we have developed, for fruitful discussions in the theoretical development, and for the overall company that he has provided during the time we worked together.

I am grateful for the assistance and support given by Tapio Leipälä, Miki Sirola, and the whole technical staff at the Department of Information and Computer Science. I also acknowledge the use of computational resources provided by Aalto Science-IT project for the run time experiments carried out in the project. I further wish to thank my officemate Srikrishna Raam for some helpful tips in language use, and my friend Ehsan Amid, for a hint on data-plotting of the run time results presented in this thesis.

Last but not least, I wish to express my deepest gratitude to my parents for rearing and nurturing me over the past 25 years. I dedicate this thesis to my beloved big brother Mubarek, and my dearest father Nesru.

Espoo, Jan 3, 2014

Abdulmelik Mohammed

Contents

Chapter 1

Introduction

In 1959, the renowned theoretical physicist, Richard P. Feynman [26] gave a famous talk to the American Physical Society where he remarked that "*there is plenty of room at the bottom*"—enough room that one can put the writing of a volume of the Encyclopaedia Britannica on the head of a pin. Feynman [26] further envisioned the miniaturization of the computers of his day. The rapid miniaturization of computing machines, codified in Moore's law, is now evident with the profusion of ever smaller electrical devices with greater capability. But, how much room is still there at the bottom, and more significantly, how much control do we have over what lies in this realm?

There is a limit to how small our computers, or in fact any machine, can get. Any material must be composed of atoms which have sizes on the scale of one billionth of a meter—a nanometer. Atoms, molecules and molecular complexes are all in the realm of the nanoscale which ranges from 0.1 nm to 1000 nm [64]. *Nanotechnology* is an engineering and scientific discipline which concerns itself with the rational design, fabrication and characterization of artificial systems at the nanoscale [64].

There are two fundamental approaches in nanoscale fabrication: a top-down approach and a bottom-up approach [11]. The top-down approach, of which the photo-lithographic technique in the semiconductor industry is a prime example, starts from a bulk material and carves out smaller details until the desired product is obtained. The bottom-up approach on the other hand, much inspired by natural systems, involves the construction of larger aggregates by the self-assembly of smaller subunits via their autonomous interactions. For scales less than 20 nm, top-down approaches can become prohibitively expensive, while bottom-up approaches mimicking nature are promising [45].

Self-assembly is a prevalent phenomenon in nature as a whole and in biology in particular. Molecular self-assembly governs the folding of linear

chain protein to functional 3D structures; protein-nucleic-acid aggregates regulate synthesis in a cell; cells divide and further organize into organisms [12]. Of the possible biological molecules that could serve as units in self-assembly, deoxyribonucleic acid (DNA), the carrier of the genetic code of life, has been demonstrated as one of the most promising in the controlled design of nanoscale structures [25]. Nanotechnology driven by the self-assembly of DNA has been termed as *DNA nanotechnology*; when the final assembled structure is static, the field has been named as *structural DNA nanotechnology*.

1.1 Problem statement

Structural DNA nanotechnology was pioneered by Nadrian Seeman in the early 1980s when he envisioned the possibility of constructing 3D crystals to host proteins which can then be studied under x-ray diffraction [49]. The field has since grown and DNA based objects [16, 65], arrays [63], nanomechanical devices [57] and even DNA molecular computers [62] have been constructed [50].

In structural DNA nanotechnology, a target can be assembled in different ways. In a multi-stranded approach, the target is entirely assembled from short single strands of DNA [47]. In an origami based approach, the majority of the target structure's mass is one long strand of DNA [47]. *Scaffolded DNA origami* is an origami based approach introduced by Paul Rothemund in mid 2000s where a single long viral strand called a *scaffold strand* is folded with the help of numerous short synthetic strands called *staple strands* [46]. Scaffolded DNA origami enables the assembly of complex custom shapes as well as the generation of complex patterns even though it forgoes some of the careful practices of the earlier multi-stranded approach [46].

In Rothemund's scaffolded DNA origami, the target shape is first geometrically modelled in a multi-step design process [46]. To approximate the shape, the scaffold strand is routed back and forth in a raster-fill manner in one of the design steps. Due to the ease of the scaffold routing in 2D origami, the scaffold route was hand-generated [45]. In general however, there will be a need for powerful computer-aided design (CAD) programs for the construction of increasingly complex DNA nanostructures [52]. Other researchers have since developed software tools for DNA origami designs [2, 22].

In a recent work, Högberg and co-workers have proposed a novel *scaffold routing* scheme in a DNA origami design adapted to polyhedral beam-frameworks [5, 36]. In their scheme, the scaffold strand is routed over the edges of a polyhedral beam-framework so that the scaffold will fold to the tar-

get polyhedron with the guide of staple strands. Scaffold routing in Högberg's team design scheme was not straightforward and their scaffold routing for a subdivided icosahedron had eight edges missing [5]. Scaffold routing plays an integral part in their vision to create a design platform for polyhedral DNA nanostructures [36] with vHelix [32]—a plugin for the well known CAD software Maya.

We started a collaboration with Högberg and his team at the Karolinksa Institutet with the role of developing an algorithm for the scaffold routing and delivering an implementation. In our work, we modelled the scaffold routing problem in graph theoretic terms and investigated the complexity of finding a suitable scaffold routing path. Furthermore, we developed a combinatorial algorithm for the proposed scaffold routing scheme, tuned in particular, to rigid 3D polyhedral beam-frameworks. In addition, we implemented a software package, BScOR (Beam Scaffolded-Origami Routing), which generates a scaffold routing path, given a three-dimensional object description in a Polygon File Format (PLY) [13]. The description of BScOR is given in the Appendix.

1.2 Structure of the thesis

The remaining chapters are organized as follows. In Chapter 2, we give a brief overview of DNA nanotechnology, focusing on DNA origami, and 3D polyhedral constructions. Further, we present the novel scaffold routing scheme proposed by Högberg's team for the DNA origami design of polyhedral beam-frameworks.

In Chapter 3, we discuss some basic graph-theoretic concepts, focusing on planarity and connectivity. We also present a relation between planarity and connectivity on the one hand and polyhedral beam-frameworks on the other. We further observe that scaffold routing on rigid 3D polyhedral beam-frameworks is equivalent to scaffold routing on a special class of planar graphs called *plane triangulations*.

In Chapter 4, we frame the question of scaffold routing for origami constructions in terms of the graph-theoretic concept of Eulerian trails. In particular, we present a special type of Eulerian trails called *A-trails*, which coincide with the notion of scaffold routing. Further, the complexity of deciding the existence of A-trails on polyhedral graphs and Eulerian plane triangulations is investigated. Deciding the existence of A-trails on general polyhedral graphs is found to be NP-complete,

Given the NP-completeness result, in Chapter 5, we describe a backtracking search algorithm for finding A-trails. To improve the performance of our

search algorithm, we then formulate a simple heuristic to aid the backtrack search. Finally, we study the performance of the heuristic by run-time experiments on three different families of graphs. In Chapter 6, we review our results and make some concluding remarks.

Chapter 2

DNA nanotechnology

There is indeed plenty of room at the bottom; while there may not yet be encyclopaedic writings at the nanoscale, the genetic information that is the essence of all life on earth, is found as a DNA sequence inside the nucleus of a cell. Such has been the impact of understanding DNA in biology and the life sciences in general, its double helical variant, depicted in Figure 2.1, has become one of the prominent symbols of science.

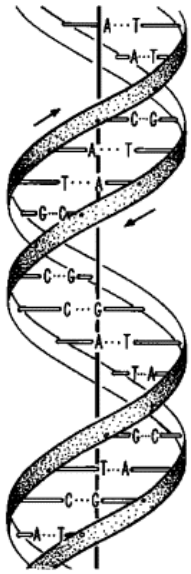


Figure 2.1: Helical structure of B-DNA. Image reprinted from [6].

DNA has multiple naturally occurring conformations; the helical geometry of B-DNA, the predominant conformation in eukaryotic cells [43], is shown in Figure 2.1. Double helical DNA consists of two single *strands* spiralling

around a helical axis. Each single strand is a chain of nucleotides which themselves are composed of a deoxyribose sugar, a phosphate group, and one of four nitrogen bases. The strands run in opposite directions from a $5'$ end to a $3'$ end, as indicated by small arrows in Figure 2.1, and are said to be *antiparallel*. The double helix is held together by hydrogen bonds between complementary nitrogen bases—Adenine-Thymine (A-T) and Cytosine-Guanine (C-G)—also named as *Watson-Crick base pairs* (bp). Under normal conditions, B-DNA has a 2 nm wide helix (in diameter) with a helical repeat (a full turn) at 10–10.5 base pairs or approximately 3.5 nm [50]. This makes DNA an intrinsic nanoscale object and DNA based constructions nanoscale products.

The string of bases in the strands is not only the code of heredity, but also a mechanism for programmable self-assembly. It is the high specificity of the hydrogen bond interaction between the nitrogen bases that enables the rational design of structures at the nanoscale [50]. For instance, if two double-stranded molecules have single-stranded antiparallel complementary overhangs (or "sticky ends"), they can hybridize to form a longer double-stranded molecule by attaching at the sticky ends through Watson-Crick base pairing [51]. Moreover, the structure formed at the location of association is still B-DNA [42] which makes the formed structure predictable [51]. In general, a target structure can be synthesized in a lab by designing sequences which maximize Watson-Crick base pairing of the structure [51].

2.1 DNA as a building material

The association of linear duplexes through sticky ends can only form topologically linear or circular products since the axis of double helical DNA is unbranched [50]. On the other hand, branched DNA molecules have more potential in creating complex shapes and features [50]. The Holliday junction, a branched DNA molecular complex with four arms, is present in biology as a transient intermediate in genetic recombination [50]. The Holliday junction has a mobile junction point because of the homology of the sequence of the original double helices; that is, the four strands of the two helices consist of two pairs of strands with the same sequence [50]. It is also possible to create synthetic branched DNA by the reciprocal exchange of strands of two double helix molecules [51]. Immobile branched junctions can be synthesized by appropriate sequence design techniques such as sequence symmetry minimization [51].

By extending the four arms of an immobile Holliday junction with sticky ends, a motif which can self assemble to a 2D periodic array can be obtained

[50]. A four arm Holliday junction with sticky ends is illustrated on the left in Figure 2.2 and a quadrilateral assembly, which can further be extended, is shown on the right. Two-dimensional DNA arrays can potentially be used to assemble molecular electronic devices [44]. Three-dimensional crystal cages based on periodic arrays were first envisioned by Seeman to host biological macromolecules for diffraction analysis [50]. While it would seem that immobile junctions are the desired building blocks for nano-constructions, four arm branched motifs with single helical arms are floppy and the junction geometry of four arm branched motifs is uncertain [47].

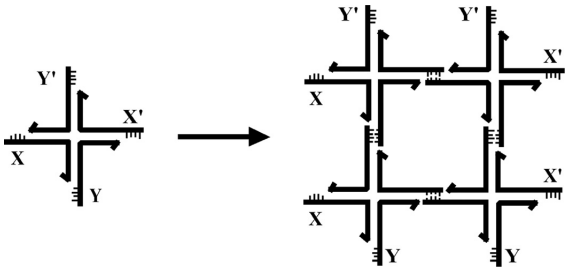


Figure 2.2: Two-dimensional array by sticky end association of branched DNA molecules. The branched junction on the left has four strands X, Y, X' and Y' and four arms corresponding to four double helical domains. Four copies of the motif associate by complementary sticky ends (X with X' and Y with Y'); the array can extended by additional associations of extra copies to the free sticky ends at the periphery. Image reprinted from [50] with kind permission from Springer Science and Business Media.

The four arms of the immobile junction have been shown to stack and form two helical domains, with two strands crossing over between the domains, and the other two strands running continuously in the two separate domains [28]. A motif with two crossover points called a *double crossover molecule* has been modelled [28], and its antiparallel variant has been shown to be approximately twice as rigid as linear duplex DNA [48]. Two-dimensional crystals based on synthetic double crossover molecules have been assembled in the lab [63], and remarkably, Winfree [62] has shown that the self-assembly of double crossover molecules can simulate computation by serving as molecular representation of so called Wang tiles.

Generally, various non-natural DNA motifs can be designed by the reciprocal exchange of strands, and sequences can be assigned to strands so that the desired motif can be assembled [51]. Complex structures in two and three dimensions can then be formed by the sticky end cohesion of the designed motifs. Nevertheless, we now turn our attention to a design technique that

has allowed the creation of arbitrary 2D shapes and patterns, and that which has been extended to 3D polyhedral architectures by Högberg's team.

2.2 DNA origami

Scaffolded DNA origami is a design strategy in DNA nanotechnology which works by the folding of a long single strand called a *scaffold strand* with the help of hundreds of shorter strands named *staple strands* [46]. In scaffolded DNA origami, the staple strand sequences are designed as Watson-Crick complements of two or more subsequences in the scaffold, thereby controlling the folding of the scaffold in a solution. Once a scaffold and a set of staples are designed and sequenced for a target shape, they are mixed in one pot with buffer and salt [45]. Next, the mixture is rapidly heated, and then cooled over a course of few hours, to get the desired shape in a solution [45].

The original DNA origami design proposed by Rothemund [46] is a multi-step process; here we will give an overview of the steps which are relevant to our work. First, the target shape is approximated by a set of paired parallel cylinders running from top down. Each cylinder models a double helix. A simple geometric shape approximated by six paired parallel helices is shown in Figure 2.3(a). Moreover, periodic crossover points are formed between the parallel helices; these crossover points are where the staples jump from one helix to the next so that the helices are held tight. Second, the scaffold is routed in the helices in a raster fill fashion, crossing over to further helices at scaffold cross over points, so that the scaffold constitutes one of the strands in each helix. The raster fill routing of a scaffold for the shape in Figure 2.3(a) is shown in Figure 2.3(b). The scaffold crossover locations are chosen so that the twist of the scaffold is close to the tangent between the helices; that is, an odd number of half turns when the scaffold progresses to subsequent helical domains and an even number of half turns when the scaffold reverses direction. Further steps are carried out to minimize strain and improve the binding strength of the staples.

Rothemund illustrated the working of his origami design principle by folding a circular single strand DNA of the M13mp18 virus (with 7,249 nucleotides) into squares, rectangles, stars, smiley faces and hollow triangles. The scaffold routing approximating the targets and microscopic images of the assembled shapes are shown in Figure 2.4. He further showed atomic force micrograph images of DNA artwork such as the map of the western hemisphere, by decorating a rectangle with a set of labelled staple strands. However, all his constructions were two-dimensional.

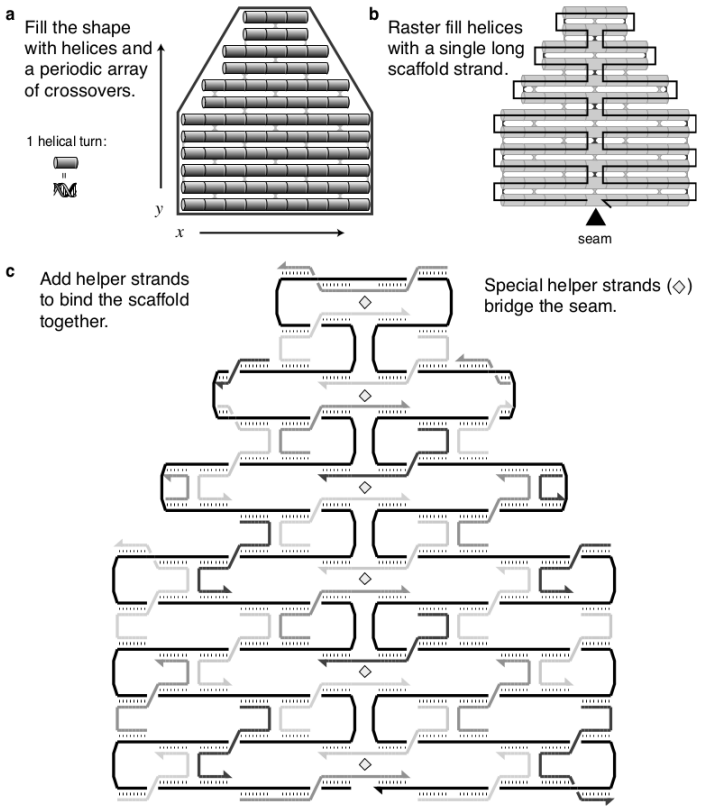


Figure 2.3: DNA origami design steps. Bold lines in b and c represent the scaffold strand. Lighter grey short strands are the staples. Reprinted from [47] with kind permission from Springer Science and Business Media.

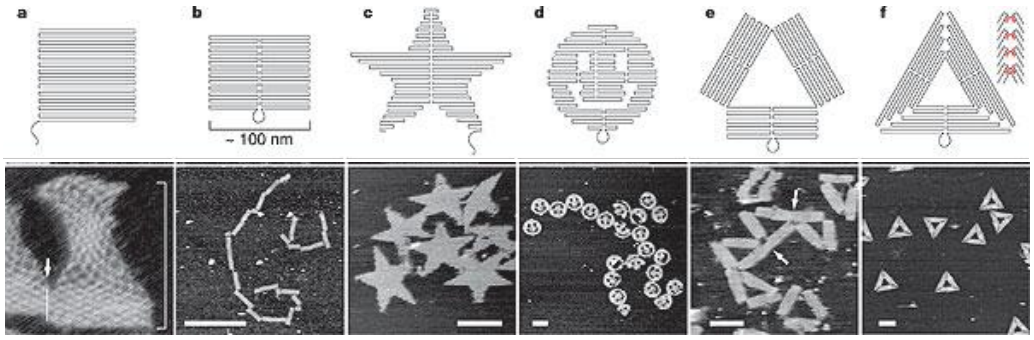


Figure 2.4: DNA origami. Top: scaffold routing path for the various shapes. Bottom: atomic force micrograph images of the folded shapes. Reprinted by permission from Macmillan Publishers Ltd: Nature [46], copyright 2006.

Since then, DNA origami has been extended to the third dimension. Anderson et al. [3] have used DNA origami to fold the M13mp18 viral DNA to a hollow 3D DNA box 42x36x36 nm^3 in size. The box consisted of six sheets corresponding to the six faces of the box where each sheet is folded in the same raster fill manner as the original Rothemund constructions. Douglas et al. [21] have used origami to construct various 3D volumes by rolling sheets of parallel helices in a honeycomb pleat based strategy. Douglas et al. [22] have developed caDNAno, an open source graphical software package, to aid the design of DNA 3D volumes constrained in a honeycomb lattice. In addition, Castro et al. [15] have developed CanDo, a computational tool for predicting 3D DNA origami shapes based on caDNAno designs.

2.3 DNA polyhedral beam-frameworks

Three-dimensional DNA nanostructures can be used for encapsulation of drugs, proteins and other nanomaterials [40]. After encapsulation, drugs can be released towards targeted cells; proteins can be controllably folded; biomaterials can be sensed or crystallized [40]. For instance, Bhatia and coworkers [10] have demonstrated a gold nanoparticle trapped in an icosahedral DNA nano-capsule. In this regard, polyhedral beam-frame architectures have the potential to create structures with a more efficient use of material (in terms of strand base pairs) and with a greater strength to weight ratio [36].

The first DNA polyhedral beam-framework was constructed by Chen and Seeman [16], and had the connectivity (or topology) of a cube. Each edge of the cube-like framework was made of double helical DNA and each vertex was a branch point of a junction. Unlike origami, which is a one pot reaction, the cube was constructed over five steps involving purification, reconstitution and ligation. Zhang and Seeman [65] have also synthesized, on solid support, a truncated octahedron with four arm branched junction (each vertex had an exocyclic arm). The synthesis of both the cube and octahedron had low yields (around 1% for the cube and less than 10% for the octahedron) [65]. Goodman et al. [29] have demonstrated a single-step synthesis of a rigid tetrahedral framework which can potentially serve as a 3D geometric building block. In their synthetic scheme, the tetrahedron was composed of four 55 bp single strands each of which corresponded to the four face boundaries of the object.

He et al. [34] have assembled three different symmetric supramolecular polyhedral frameworks (tetrahedra, dodecahedra and buckyballs) from sticky-ended three-point-star motifs in a hierarchical strategy. Joyce et al.

[53] were the first to use an origami technique[1] to fold an octahedral beam-framework from a 1.7 kilo-base single stranded synthetic DNA molecule. Douglas et al. [21] have assembled an icosahedral beam-framework from three origami based double-triangular subunits in a hierarchical strategy. Each of the edges of the beam-framework were six helix bundles.

Recently, Högberg and his team [5] proposed an origami design technique for a subdivided icosahedral framework. A scaffold strand was routed through the edges of the subdivided icosahedron so that it did not cross itself at any vertex. Then, staple strands were used to form a vertex by acting as complements to the scaffold running in an antiparallel direction. Vertex formation is illustrated in Figure 2.5. The routed scaffold had eight edges missing and the missing edges were compensated for by complementary pairs of staples. The resulting icosahedral design is depicted in Figure 2.6.

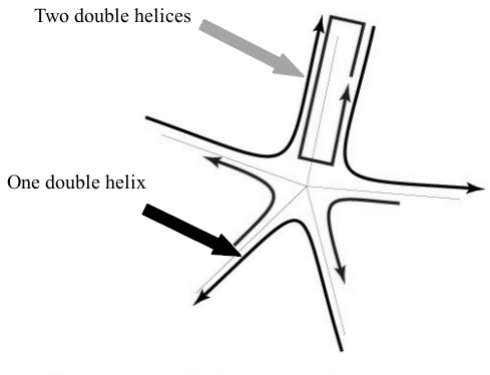


Figure 2.5: Vertex formation according to Högberg's origami design for a subdivided icosahedron. Thin grey lines are edges of the subdivided icosahedron. The longer bold lines are segments of the routed scaffold at the vertex; the shorter bold lines are the staples. Arrows indicate the $5'$ to $3'$ direction.

In the work presented in this thesis, we have formalized this design scheme in graph theoretic terms. We then investigated the conditions under which the scaffold can be routed for arbitrary polyhedral beam-frameworks. Moreover, we have designed an algorithm which outputs a scaffold routing path for a polyhedral beam-framework if such a path exists. We have paid particular attention to the class of rigid polyhedral beam-frameworks. We have implemented a software package, BScOR, which outputs a scaffold routing path for an arbitrary polyhedral framework if such a routing path can be found. A description of BScOR is available in the Appendix.

[1]The origami technique by Joyce et al. precedes Rothemund's scaffolded origami and uses a single strand with only five helpers.

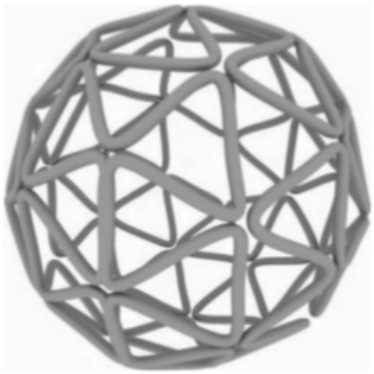


Figure 2.6: Scaffold routing over the subdivided icosahedron with some missing edges. The single helical or double helical domains made by the hybridization of the scaffold with the complementary staples act as beams of the DNA nanostructure. Reprinted with permission from [36].

Chapter 3

Graph theoretic preliminaries

Graph theory is a powerful abstraction tool to study various types of networks such as road networks, telephone networks, and the World Wide Web. While these networks in a certain sense seem natural objects to study using graph theoretic tools, it will be evident from the discussions that follow that graph theory can also be powerful in modelling problems that arise in DNA nanotechnology. Let us first define the basic graph theory terms and the notations that we shall use in this thesis.

A *graph* is a pair $\langle V, E\rangle$, where V is a set of *vertices* (also called *nodes*) and E is a multiset of unordered pairs of vertices called *edges*. We denote an edge $e \in E$ as a pair $\{u, v\}$, where $u, v \in V$, and we say u and v are *endpoints* of e. We denote by $V(G)$ and $E(G)$, the vertex set and edge set of a graph G, respectively. In this work, we consider finite, undirected graphs which may have *multiedges* (multiple edges with the same endpoints) but which may not contain any loops (edge from a vertex to itself). We say two edges are *parallel* if they have the same endpoints. We call a graph *simple* if it does not contain multiedges.

An edge is said to be *incident* to a vertex if the vertex is one of the endpoints of the edge. The *degree* $deg(v)$ of a vertex v, is the number of edges incident to v in the graph. A vertex u is *adjacent* to a vertex v if $\{u, v\}$ is an edge in the graph. The vertices adjacent to a vertex v are called the *neighbours* of v. If the degree is k for all vertices, we say the graph is *k-regular*. We call a 3-regular graph *cubic*.

A *walk* in a graph is an alternating sequence of vertices and edges $(v_0, e_1, v_1, \ldots, e_l, v_l)$, $l \geq 0$, where each $e_i = \{v_{i-1}, v_i\}$ is an edge in the graph. For clarity, we will ignore the edges in the sequence if no ambiguity arises (e.g. if the graph is simple). A walk is *closed* if it starts and ends in the same vertex; otherwise, it is *open*. An open walk with distinct vertices (that is, no vertex is repeated) is called a *path*. The start and end vertices of a

path are its *end vertices* while the remaining are its *internal vertices*. A walk with distinct edges is called a *trail* and a closed walk with distinct edges is a *closed trail*. Note that parallel edges are considered distinct. A *cycle* is a closed trail with at least two vertices where no vertex (except the start vertex) is repeated. The length of a walk (path, trail, closed trail, cycle) is the number of edges in it.

A graph is *connected* if there is a path between any pair of its vertices; otherwise, it is *disconnected*. Unless stated otherwise, we assume our graphs are connected. In Section 3.2, we discuss connectivity in more general terms.

A graph G is called *bipartite* if $V(G)$ is the union of two disjoint, possibly empty, sets such that no edge $e \in E(G)$ has both endpoints in the same set. A *complete graph* is a simple graph where all the vertices are pairwise adjacent. We denote a complete graph on n vertices by K_n. A *complete bipartite graph* is a simple bipartite graph where any two vertices are adjacent if and only if they are in two different partite sets of its vertex set. A complete bipartite graph with two partite sets of size m and n is denoted $K_{m,n}$. For other undefined terms, we refer the reader to the introductory book on graph theory by Douglas West [59].

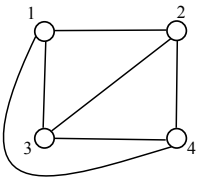
In the next two sections, we present the graph-theoretical concepts of planarity and connectivity which are then shown to have a relation with polyhedral frameworks.

3.1 Planarity

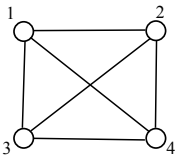
A graph can be drawn on a plane by injectively mapping its vertices to points on the plane and its edges to continuous curves connecting their endpoints. If a drawing of a graph on a plane is such that no two curves corresponding to the edges intersect apart from their endpoints, the drawing is said to be *planar*. If a graph has a planar drawing, then it is called a *planar graph* [60]. Figure 3.1 illustrates two possible drawings of K_4, the complete graph on four vertices. The drawing of K_4 in Figure 3.1(a) is a planar drawing, while the drawing in Figure 3.1(b) is non-planar since there is an intersection between edges $\{1, 4\}$ and $\{2, 3\}$. The image of edge $\{1, 4\}$ in the Figure 3.1(a) is not straight. Fáry's theorem [59, p.247] states that any simple planar graph[1] has a straight line planar drawing (that is, the images of all edges are straight lines).

In general, a planar graph may have multiple planar drawings. If one simply considers different mappings (i.e. different geometric instantiations)

[1]Note that a planar graph with at least one multiedge cannot have a straight line planar drawing.



(a) Planar drawing of K_4.



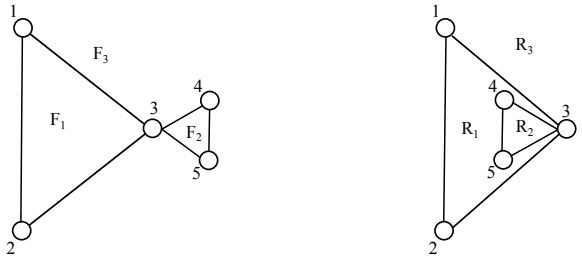
(b) Non-planar drawing of K_4.

Figure 3.1: Drawings of K_4.

to distinguish planar drawings, then any planar graph will have an infinite number of drawings (any rubber sheet transformation of a particular drawing is defined by a different mapping). However, a planar drawing defines, for each vertex, a cyclic order (e.g. counter-clockwise) of the edges incident to that vertex [8].

Thus, we consider two planar drawings to be equivalent if they define the same cyclic order for each vertex. We consider two planar drawings to be different if one needs to rearrange the order of the edges around at least one of the vertices of first drawing to obtain the second drawing. The class of all equivalent planar drawings defines a *combinatorial embedding* of the graph. Figure 3.2 displays two different combinatorial embeddings (hereafter also just embedding) of an abstract planar graph with a vertex set, $V = \{1, 2, 3, 4, 5\}$ and an edge set $E = \{\{1, 2\}, \{1, 3\}, \{3, 4\}, \{3, 5\}, \{4, 5\}\}$. If we want to obtain the embedding in Figure 3.2(b) from the embedding in Figure 3.2(a), we would have to fold the bow-tie embedding in Figure 3.2(a) at a vertical line passing through the centre at vertex 3. This rearranges the cyclic order of the edges at vertex 3 from $(\{1, 3\}, \{2, 3\}, \{5, 3\}, \{4, 3\})$ to $(\{1, 3\}, \{4, 3\}, \{5, 3\}, \{2, 3\})$. A *plane graph* is the class of all equivalent planar drawings; that is, the class of planar drawings which define the same combinatorial embedding. We will abuse notation and use G to denote plane graphs; in such a case, a cyclic order of the edges incident to a vertex is assumed to be given for each vertex.

Now, suppose that we remove the curves of a plane graph from the plane: the plane becomes divided into disjoint regions [8]. These regions are the *faces* of the plane graph [8]. For instance, the bow-tie embedding in Figure 3.2(a) has 3 faces: the triangles F_1, F_2; and the unbounded outer face F_3. Each face of a plane graph is *bounded* by a closed walk (but not necessarily a cycle) [59]. The face F_1 is bounded by $(1, 2, 3, 1)$, F_2 by $(3, 5, 4, 3)$ and F_3 by $(1, 2, 3, 5, 4, 3, 1)$. On the other hand, the embedding in Figure 3.2(b) has three faces R_1, R_2 and R_3 with bounding walks $(1, 2, 3, 5, 4, 3, 1)$,



(a) Bow-tie embedding.

(b) Half double-bow-tie embedding.

Figure 3.2: Two different combinatorial embeddings of a planar graph. The two embeddings correspond to two plane graphs.

$(3, 4, 5, 3)$, $(3, 1, 2, 3)$, respectively. The *length of a face* is the length of the walk bounding the face: F_1 and F_2 have length 3, while F_3 has length 6.

In general, the number of faces of a plane graph is determined by the number of vertices and edges of a graph in accordance with a classical theorem by Euler (Theorem 3.1.1). A proof of the theorem by induction can be found in most graph theory textbooks (e.g. [37, p.22]).

Theorem 3.1.1 (Euler)**.** *Let $G = \langle V, E\rangle$ be a plane graph and let F denote the set of faces of G. Then, $|V| + |F| = |E| + 2$.*

Since $|V|$ and $|E|$ are independent of the embedding, any two embeddings of a planar graph have the same number of faces. A direct consequence of Euler's theorem is a bound on the number of edges of a simple planar graph [59, p.241].

Theorem 3.1.2. *Let $G = \langle V, E\rangle$ be a simple planar graph with at least three vertices. Then, $|E| \leq 3|V| - 6$.*

Proof. We derive the bound by counting the number of edge-face pairs in an arbitrary embedding of G with face set F. Each edge of G bounds at most two faces. Thus, the number of edge-face pairs is at most $2|E|$. Since G is a simple graph, the length of any face in F is at least three. Thus, the number of edge-face pairs is at least $3|F|$; hence, $3|F| \leq 2|E|$. Substituting $|F| = |E|+2-|V|$ from Euler's Formula (Theorem 3.1.1), we obtain $|E| \leq 3|V| - 6$. □

We note following details in Theorem 3.1.2. We can construct a planar graph with an unbounded number of edges by drawing multiedges as dense fibres along the curve of an edge; however, such a graph would not be simple.

Moreover, two parallel edges of a graph can form a face of length 2. Further, the only simple graph with two vertices has one edge while the bound suggests zero edges. Similarly, an isolated vertex has zero edges while the bound would be negative.

Theorem 3.1.2 states that simple planar graphs are sparse, in the sense that they do not have many edges. From the complexity point of view, it implies that $|E| = \mathcal{O}(|V|)$ for a simple planar graph. Hence, algorithms such as breadth first search which take $\mathcal{O}(|V| + |E|)$ steps, will take $\mathcal{O}(|V|)$ steps for simple planar graphs.

A simple plane graph where all the faces, including the outer face, are 3-cycles is called a *plane triangulation* [59]. If a simple plane graph has faces of length 4 or more, we can add more edges within these faces until all faces in the graph become 3-cycles. Once all the faces are 3-cycles, we cannot add more edges without losing planarity unless the added edges are multiedges. Thus, plane triangulations are *maximally plane simple graphs* [59, p.242]. From the proof of Theorem 3.1.2, we can deduce that a plane triangulation has exactly $3|V| - 6$ edges [59]. Further, no other type of simple planar graphs have this many edges. Thus, we can easily test whether a simple planar graph is a plane triangulation by counting the number of edges.

Faces of a plane graph resemble territorial regions of a geographic map. In constructing a graph from a geographic map, we place vertices in each territory and connect two vertices if the corresponding regions share a boundary. What kind of graph would we obtain if we made a similar construction from a plane graph? The (geometric) *dual* of a plane graph G, denoted G^*, is the plane graph obtained by:

1. Placing exactly one vertex on the interior of each face of G,
2. Connecting vertex v_1 and $v_2 \in V(G^*)$ by one arc through the edge shared with faces of G corresponding to v_1 and v_2.

Figure 3.3 depicts the dual graph of the bow-tie embedding in Figure 3.2(a): the rectangles are the vertices and the dotted curves are the edges. Here, the dual is a plane graph with 3 vertices and 6 edges with multiple edges between two pairs of vertices. Note that even when a plane graph is loopless, its dual is not necessarily so—consider the case where G contains a degree one vertex [8]. Nonetheless, the class of graphs that are relevant in our considerations[2] will have loopless duals.

By construction, if G^* is the dual of G, then $|V(G^*)| = |F(G)|$, $|E(G^*)| = |E(G^*)|$ and $|F(G^*)| = |V(G)|$. Moreover, face lengths in G correspond to

[2]Our graphs will be at least 2-connected. In Section 3.2, we will define 2-connectedness.

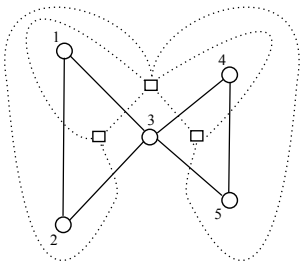


Figure 3.3: The dual of a plane graph.

vertex degrees in G^*. It is also easy to see that the dual of the dual of a connected plane graph is isomorphic to the original graph. Hence, face lengths in G^* correspond to vertex degrees in G. However, two distinct combinatorial embeddings of a planar graph may have non-isomorphic duals [59].

Let us now return to the notion of planarity. What type of graphs are planar? The complete graph on five vertices, K_5, has 10 edges—above the bound imposed by Theorem 3.1.2—and thus, is not planar. Another minimal non-planar graph (in the number of edges) derives from the puzzle of trying to connect three utilities, water, electricity and gas, to three cottages without an intersection between the connections [41]. This graph is the complete bipartite graph $K_{3,3}$. In fact, the relationship between these two graphs and planarity goes deeper and is expressed by Kuratowski's theorem (Theorem 3.1.3) [59]. To state Kuratowski's theorem, we first need to define the notion of a subdivision of a graph.

A *subdivision* of an edge $e = \{u, v\}$ is an operation which adds an intermediate vertex w between the endpoints of e so that $\{u, v\}$ is replaced by two edges (u, w) and (w, v). A graph H is said to be a *subdivision* of a graph G if it can be iteratively constructed by the subdivision of edges. A subgraph of a graph which is a subdivision of K_5 or $K_{3,3}$ is called a *Kuratowski subgraph*. We can now state the relationship between planarity and Kuratowski subgraphs.

Theorem 3.1.3 (Kuratowski)**.** *A graph is planar if and only if it does not contain a Kuratowski subgraph.*

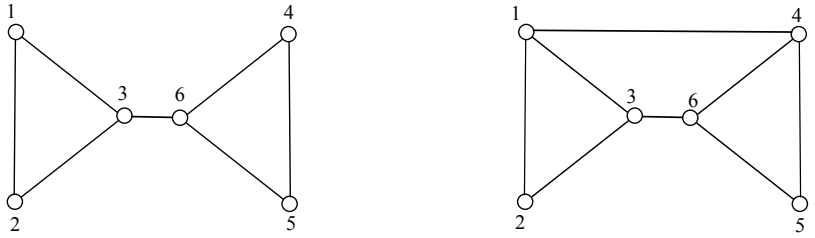
The necessity condition for planarity is easy to prove. Indeed, a graph containing a Kuratowski subgraph had a planar drawing, we could restrict the drawing of the graph to the Kuratowski subgraph, erase the subdivision vertices, and obtain a planar drawing of K_5 or $K_{3,3}$. This is a contradiction and thus the graph could not have been planar. The proof of the sufficiency

is more involved and thus omitted for conciseness. The reader can find it in most graph theory textbooks (e.g [59, p.261]).

Kuratowski's theorem gives a complete characterization of planar graphs. However, it is not easy to obtain an efficient planarity testing algorithm from Kuratowski's theorem as a direct test for the existence of a Kuratowski subgraph entails a minimum cost of $\mathcal{O}(|V|^6)$ [35]. In our software package BScOR, we used an $\mathcal{O}(|V|)$ planarity testing and embedding algorithm by Boyer and Myrvold [14] as implemented in the boost graph library for `C++` [61] (cf. the fifth executable in the BScOR pipeline in the Appendix).

3.2 Connectivity

Graph connectivity is another central concept of graph theory with several applications. We briefly mentioned graph connectivity previously, and we now present its generalization. A graph with at least $k+1$ vertices is *k-connected* if the removal of any $k-1$ or fewer vertices does not leave the graph disconnected. As such, a graph is connected if it is 1-connected. A vertex whose removal disconnects a graph is called a *cut-vertex*. By definition, a 2-*connected graph* has no cut-vertex. The graph in Figure 3.4(a) is not 2-connected because vertex 3 is a cut-vertex; while the graph in Figure 3.4(b) is 2-connected. Similarly, a *cut-pair* is a vertex pair whose removal disconnects a graph. The set $\{1, 3\}$ is a cut-pair for both graphs in Figure 3.4. A 3-*connected* graph does not contain a cut-pair or a cut-vertex. Both graphs in Figure 3.4 are not 3-connected. By convention, the complete graph on n vertices, K_n, is not n-connected [59]. Moreover, the connected graph on two vertices is considered 1-connected but not 2-connected [59].



(a) A connected graph. (b) A 2-connected graph.

Figure 3.4: k-connectedness of graphs.

Analogously, a graph is *k-edge-connected* if the removal of any $k-1$ edges does not disconnect the graph. For instance, the graph in Figure 3.4(b) is 2-edge-connected, while the graph in Figure 3.4(a) is not. A *bridge* is then

the analogue of a cut-vertex: it is an edge whose removal disconnects a graph. Edge $\{3, 6\}$ in Figure 3.4(a) is a bridge. A *disconnecting-set* is a set of edges whose removal leaves a graph disconnected. If the removal of k edges disconnects a graph, the removal of some k endpoints of the edges would disconnect the graph as well. Therefore, a k-connected graph is also k-edge connected [59].

In a connected graph, there is a path between any two vertices. Suppose a connected graph G is not 2-connected and has a cut-vertex v. When v is removed, we have at least two components, say C_1 and C_2. Any path in G from a vertex u in C_1 to a vertex w in C_2 must pass through v; otherwise the path from u to w without v would connect C_1 and C_2. Hence, all paths from u to w contain v and are not disjoint. Formally, a path P_1 from x to y is *internally disjoint* with another path P_2 from x to y if all the internal vertices of P_1 are different from the internal vertices of P_2. In the graph G, there are no internally disjoint paths between u and w.

What if a graph is 2-connected? Would it guarantee that we have two internally disjoint paths between any two vertices? Whitney's theorem (Theorem 3.2.1) states that it is indeed sufficient that a graph is 2-connected for it to have two internally disjoint paths between any two of its vertices [59].

Theorem 3.2.1 (Whitney)**.** *A graph with at least three vertices is 2-connected if and only if there exist at least two internally disjoint paths for any two vertices in the graph.*

In Section 3.1, we mentioned that the dual of a plane graph may have loops. For instance, the dual of the plane graph in Figure 3.4(a) has a loop due to the bridge edge $\{3, 6\}$. The edge only bounds one face. In contrast, all the edges in the 2-connected plane graph in Figure 3.4(b) bound exactly two faces and hence the dual of the graph is loopless. In fact, 2-connectedness is sufficient for a plane graph to have a loopless dual.

Lemma 3.2.2. *An edge of a 2-(edge)-connected plane graph bounds exactly two faces. Hence, the dual of a 2-(edge)-connected plane graph is loopless.*

Proof. Any edge e of a 2-edge-connected graph is contained in some cycle: the two endpoints of the edge must have at least one more path for otherwise e would be a bridge. Such a cycle would divide the plane in two regions—its interior and exterior[3]. The region on the two sides of e correspond to distinct faces as one of them must be in the interior while the other in the exterior. Thus, each edge of a 2-edge-connected plane graph bounds exactly two faces. Moreover, the dual of a 2-edge-connected plane graph is loopless. Since a

[3]This is known as the Jordan curve theorem.

2-connected graph is 2-edge-connected, 2-connected plane graphs also have loopless duals. □

If a plane graph is 3-(edge)-connected, we have a stronger restriction on the duals [7, p.229].

Lemma 3.2.3. *The dual plane graph G^* of a 3-(edge)-connected plane graph G is simple.*

Proof. Since a 3-edge-connected plane graph is also 2-edge-connected, we know G^* is loopless by Lemma 3.2.2. We now show that if G^* has multiedges, then G is not 3-(edge)-connected. Suppose G^* has multiedges. In particular, let e_1^* and e_2^* be two parallel edges in G^* with endpoints $u^*, v^* \in G^*$. The cycle $C = (u^*, e_1^*, v^*, e_2^*, u^*)$ divides the plane into two regions—its interior and exterior. Let e_1 be the edge in G that e_1^* crosses, and let e_2 be the edge in G that e_2^* crosses. The edge e_1 has an endpoint x in the interior of C and an endpoint y in the exterior of C. The removal of e_1 and e_2 disconnects x from y; this follows because only e_1 and e_2 cross e_1^* and e_2^*. Hence, $\{e_1, e_2\}$ is a disconnecting-set of G and G is not 3-(edge)-connected. □

In Section 3.1, we introduced the concept of plane triangulations which are maximally plane simple graphs with all faces as 3-cycles. Since all the faces of plane triangulations are 3-cycles, their duals are cubic (this follows because face lengths in a plane graph correspond to vertex degrees in the dual). On the other hand, since vertex degrees in a plane graph correspond to face lengths in the dual, it would seem that plane triangulations are the duals of cubic plane graphs. However, for cubic graphs which are not 3-connected, the faces in the dual may be bounded by multiedges or loops[4]. Figure 3.5 shows a cubic plane graph which is not 2-connected. The dual plane graph's vertices are represented by rectangles, and its edges are represented by dotted curves. On the other hand, if a cubic plane graph is 3-connected, then its dual is simple by Lemma 3.2.3. This gives us the following theorem which will use in Chapter 4.

Theorem 3.2.4. *The dual of a cubic 3-connected plane graph is a plane triangulation.*

It is also worth noting that the dual of a 2-connected simple plane graph is also 2-connected [20]. Likewise, the dual of a 3-connected simple plane graph is also 3-connected [20]. Moreover, a plane triangulation with at least

[4]In the definition of some authors [27], plane triangulations are allowed to have multiedges and loops. In our case, plane triangulations are simple graphs.

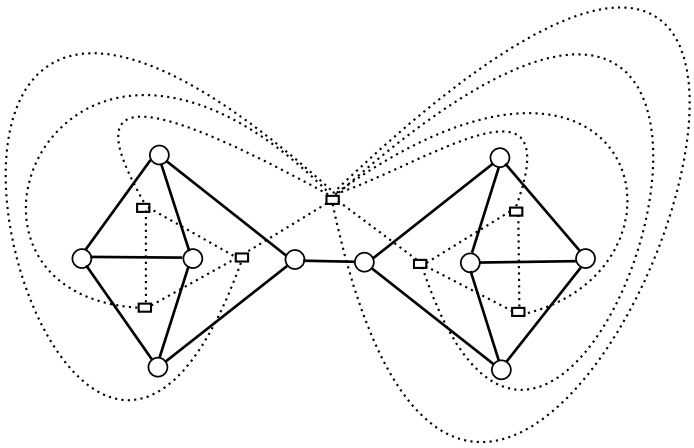


Figure 3.5: A cubic plane graph whose dual is not a plane triangulation.

four vertices is 3-connected [33, p.105]. Hence, the dual of a plane triangulation with at least four vertices is a cubic 3-connected simple plane graph. Nevertheless, it will the be the result of Theorem 3.2.4 that will be relevant to our consideration in Chapter 4. Next, we present the relationship between polyhedral frameworks on the one hand and planarity and 3-connectedness on the other hand.

3.3 Polyhedral graphs and beam-frameworks

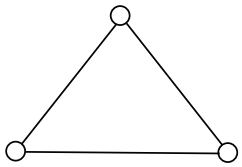
As noted in Section 2.3, we are interested in building rigid polyhedral DNA nanostructures. How do planarity and connectivity relate to such structures? The rigidity of a three-dimensional structure heavily depends on its skeleton (or framework)[5]: the points and links which constitute it [38]. In graph theoretic terms, these are the vertices and edges of an underlying graph. In this section, we describe the relation between polyhedral structures and their underlying graphs.

The DNA double helix is a stiff polymer which behaves as a rigid beam between 10 nm and 50 nm in the B-DNA conformation [40]. On the other hand, the branch points of a DNA junction are floppy [54]. We can then think of a DNA polyhedral beam-framework as a structure where the beams are rigid but the joints have rotational freedom.

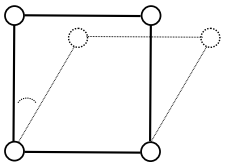
Formally, a *framework* can be defined as a simple graph geometrically instantiated in an Euclidean space [31]. The vertices assume a point in this

[5]We use the term skeleton in relation to the graph structure of a polyhedron and framework in relation to the rigidity of the polyhedral skeleton.

space and these points are connected by straight lines according to the graph adjacency; hence, graph edges have a definite length. A straight line drawing of a planar graph with fixed edge lengths and fixed angles at vertices can, for instance, be considered a framework in 2-space. A framework is *rigid* if any edge length preserving deformation of the framework will only give a framework which is congruent to it. A framework A is *congruent* with a framework B, if A is simply a translation and/or rotation of B. A framework which is not rigid is *flexible*. The triangle in Figure 3.6(a) is rigid while the square next to in Figure 3.6(b) is flexible. The square can be deformed to a set of rhombi while preserving the side lengths.



(a) A rigid triangle.

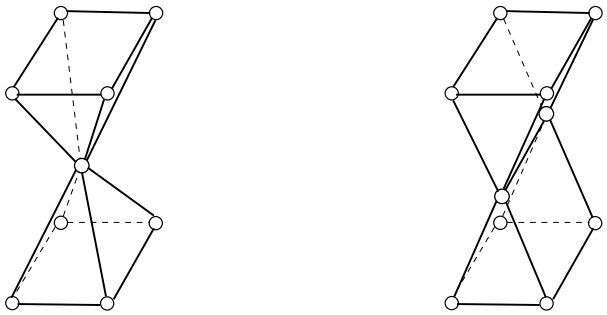


(b) A flexible square.

Figure 3.6: Rigidity of frameworks.

In simple terms, a *polyhedron* is a closed three-dimensional solid with its surface bounded by plane faces [38]. In such a definition, edges are line segments of plane intersections while vertices are points where at least three lines meet. The framework of a polyhedron comprises its edges and vertices when it is fixed in space. However, such a definition is too general and allows for flexible frameworks. The two structures shown in Figure 3.7 are polyhedra according to the simple definition but have flexible frameworks. Notice how the central vertex of double pyramid in Figure 3.7(a) is a cut-vertex of the skeletal graph; likewise, the central pair in Figure 3.7(b) is a cut-pair. The spaces enclosed by both polyhedra in the Figure are non-convex, in the sense that one can draw a line with endpoints in the enclosed spaces but with some points on the line outside the enclosed spaces. Convexity will not guarantee us rigidity but it will simplify the discussion on rigidity. We thus assume our polyhedra are convex.

Imagine the polyhedral faces are made of elastic rubber. If we puncture a hole in a face and open the rubber inside out until it becomes flat, we obtain a plane graph from the polyhedron. The plane graph obtained in such a fashion from a convex polyhedral skeleton is called a *polyhedral graph*. The face that was punctured becomes the unbounded face of the plane graph. In fact, if we blow into the punctured hole until the polyhedral surface inflates to a



(a) Two pyramids joined at their apex.

(b) Two wedges joined at a hinge.

Figure 3.7: Flexible solids.

sphere (this is always possible for convex polyhedra [19, p.57]), the polyhedral skeleton would then become an embedding of a graph on a sphere. In fact, any graph embedded on a sphere can always be embedded on a plane and vice versa [59]. Another way to obtain a polyhedral graph of a convex polyhedron is by a perspective projection of its skeleton from a point close to one of the faces, to a plane on the other side of the polyhedron [41]. Figure 3.8 depicts the polyhedral graphs corresponding to the cube, the tetrahedron and the octahedron projected from faces $(3, 7, 8, 4)$, $(1, 3, 4)$ and $(1, 4, 5)$, respectively.

We have seen how to obtain a polyhedral graph from a convex polyhedron; but, how well connected is a polyhedral graph? The relation of polyhedral graphs with 3-connected simple planar graphs is then established by Steinitz theorem (Theorem 3.3.1) [66, p.103] which we present here without proof.

Theorem 3.3.1 (Steinitz)**.** *The skeleton of a convex polyhedron is a* 3*-connected simple planar graph. Moreover, any* 3*-connected simple planar graph can be realized as a skeleton of a convex polyhedron.*

Thus, polyhedral graphs are exactly the 3-connected simple planar graphs. We can thus restate the problem of routing a scaffold for a three-dimensional polyhedral beam-frame nanostructure as one of routing a scaffold in a 3-connected simple planar graph. This allows us, for instance, to conveniently use the Jordan curve theorem in our theoretical development.

Not all polyhedral frameworks are rigid—even the convex ones. The cube, for instance, is not rigid as it easily deforms to a set of rhombic parallelepipeds by pivoting on its base. What then guarantees the framework to be rigid? It happens that the framework of a convex polyhedron with all faces triangular is rigid [18]. Clearly the convexity is necessary as, for instance, triangulating the upper and lower faces of the double pyramid in Figure 3.7(a) would

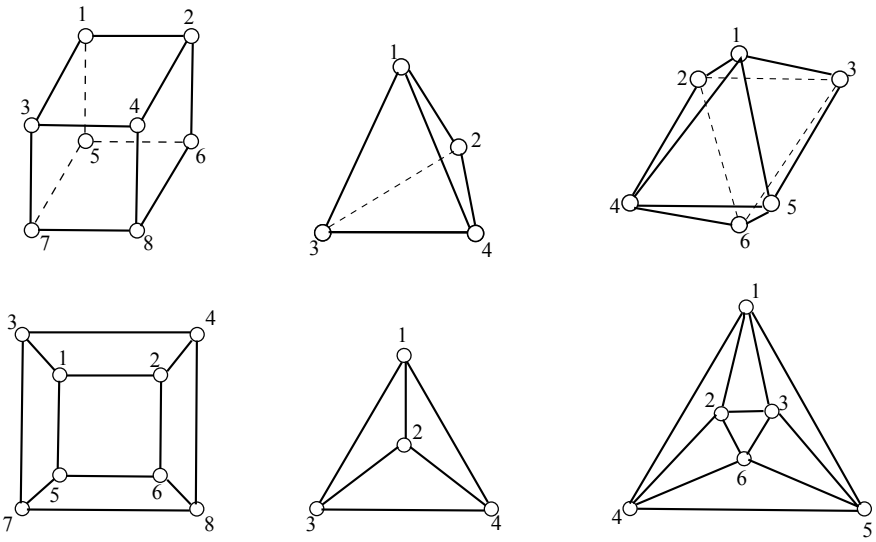


Figure 3.8: Some polyhedra and their corresponding planar graphs. On the top are the solids and on the bottom are the corresponding graphs. From left to right we have the cube, the tetrahedron and the octahedron. Projections are not according to scale.

make all faces triangular while the structure remains flexible[6]. In the case of convex polyhedra with triangular faces, the plane graph obtained by perspective projection is a plane triangulation. The octahedral and tetrahedral projections of Figure 3.8 are such plane triangulations. Hence, the problem of scaffold routing on rigid 3D polyhedral frameworks translates to the problem of scaffold routing on plane triangulations.

The intuition behind the rigidity of triangular frameworks can be explained by the fact that plane triangulations have $3|V| - 6$ edges. Each vertex as a point in 3-space would have three coordinates. For $|V|$ vertices, this amounts to $3|V|$ coordinates or $3|V|$ degrees of freedom. Since edge lengths are preserved in a deformation, each edge length reduces the degrees of freedom by one [38]. Thus, a triangular framework with $3|V| - 6$ edges has six degrees of freedom remaining. The remaining degrees of freedom are accounted for by the translational and rotational freedom of the framework itself.

[6]Nonetheless, the triangulated double pyramid will still have one non-triangular face in an embedding on a plane.

Recall that a planar graph may have two different combinatorial embeddings (e.g. the two embeddings in Figure 3.2). On the other hand, the skeleton of a convex polyhedron defines a cyclic order of edges for each vertex. It is easy to see this given that a convex polyhedron can always be inflated to a sphere [19, p.57]. For instance, the counter-clockwise cyclic order of the edges incident with vertex 3 in the 3D cube in Figure 3.8 (when inflated to a sphere) is $(\{4,3\},\{1,3\},\{7,3\})$. As will be evident from the discussion in Chapter 4, embeddings are relevant in the problem of scaffold routing. How can we obtain the order as defined in the 3D skeleton (or equivalently in an embedding in a sphere)? One possible method is to first obtain the adjacency relationship between the vertices in the polyhedral skeleton, and then run a planar embedding algorithm such as the Boyer Myrvold algorithm mentioned in Section 3.1. Would we obtain the embedding defined in the skeleton if we used such a method? To answer this, we first need to define an isomorphism between two combinatorial embeddings of a planar graph.

When we unfurl an embedding of a graph on a sphere and lays it inside out on the plane, the counter-clockwise cyclic order of edges around each vertex on the plane is a reversal of the counter-clockwise cyclic order of the edges around the vertices on the sphere. For instance, the counter-clockwise cyclic order around vertex 4 in the 3D octahedron in Figure 3.8 is $(\{1,4\},\{2,4\},\{6,4,\},\{5,4\})$, which is a reversal of the counter-clockwise cyclic order around vertex 4 in the octahedral projection in the plane. We consider an embedding O_1 of a planar graph G to be *isomorphic* with an embedding O_2 of G, if, for each vertex v in G, the cyclic order of edges around v in O_1 is simply a reversal of the cyclic order of edges around v in O_2. Note that if two edges are consecutive in a cyclic order around a vertex in an embedding, they are also consecutive in the embedding isomorphic to it.

Whitney's unique embedding theorem states that the embedding of a 3-connected simple planar graph is unique, up to isomorphism [27]. Thus, for a 3-connected simple planar graph, we can directly use an embedding we get from a planar embedding algorithm.

Chapter 4

Scaffold routing and Eulerian trails

In the origami design scheme of Högberg and his team introduced in Section 2.3, the scaffold and complementary staples form one or two helical domains which serve as the beams in the DNA polyhedral nanostructure. In Chapter 3, we presented an equivalence between polyhedral skeletons and 3-connected simple planar graphs (Theorem 3.3.1). In this chapter, we turn our attention to the problem of scaffold routing: the path the scaffold strand must follow so that all the polyhedral edges are covered, and will likely fold to the polyhedral beam-framework when stapled with antiparallel staple strands.

To avoid missing beams, the scaffold strand should be routed at least once per each edge of the polyhedron. On the other hand, for optimal use of scaffold base pairs, the routing should visit every edge at most once. From the graph-theoretic point of view, this amounts to visiting all the edges of the polyhedral graph exactly once. Is this always possible? If so, how do we find it? Leonhard Euler initiated graph theory when he puzzled the possibility of visiting the seven bridges of Königsberg exactly once. In this chapter, we state the problem of scaffold routing in terms of Eulerian trails, the same principle that Euler had used to resolve his puzzle.

4.1 Eulerian trails and postman tours

Formally, an *Eulerian trail* is a closed trail which visits every edge of a graph. For our purposes, we choose the starting vertex and the orientation of a closed trail as desired. A graph is *Eulerian*, if it admits an Eulerian trail. Our questions then become: 1) Is our graph Eulerian? 2) If it is Eulerian, can we find an Eulerian trail of the graph efficiently? The more central question that is of concern to us and that will be discussed later is whether an ordinary Eulerian trail suffices in scaffold routing. We will leave

that for now in order to discuss the basics. We first state a fundamental characterization of Eulerian graphs.

Proposition 4.1.1. *A connected graph is Eulerian if and only if all the degrees of the vertices are even.*

Proof. (if) Suppose we are given a graph G where all the vertex degrees are even. We first take an initial vertex $v_0 \in V$ arbitrarily. Next, we construct a closed trail T, by traversing the edges of G, marking them as visited, until we return to v_0. We do not visit the marked edges again. Since there are no odd degree vertices, there is always an exit edge incident to the vertex that we have visited last. Thus, our traversal cannot stop until we return to v_0. The vertices that we have visited in the traversal all have an even number of their incident edges as marked. We maintain the vertices which have been traversed but have unmarked incident edges in a list L. We then select a new vertex, u, from the list L and construct a new closed trail $T^{'}$ ending in u; marking the newly visited edges and updating L as necessary. We then modify the original trail T by splicing $T^{'}$ to T at u (that is, if (e_1, u, e_2) is a subsequence in T, we replace u with $T^{'}$ to obtain the new T). We repeat the procedure, constructing new closed trails and splicing them to T until all edges have been marked. The trail T we obtain at the end of the process is an Eulerian trail and thus G is Eulerian.

(only if) Consider an Eulerian trail of G, $T = (v_0, e_1, v_1, e_2, \ldots, e_{|E|}, v_0)$. Since we do not have any self loops[1], no vertex appears consecutively in the sequence. For every instance a vertex v appears in the sequence, it appears sandwiched between two of its incident edges. Thus, the degree of v is twice the number of times it appears in the list. □

We can thus check if a graph is Eulerian by checking that all the vertex degrees are even. Moreover, the first part of the proof gives us an algorithm for finding an Eulerian trail of an Eulerian graph. This algorithm was given by Hierholzer; a detailed discussion can be found in [37, p.42]. From the scaffold routing point of view, Proposition 4.1.1 implies that if the polyhedral graph is Eulerian, then all the beams of the DNA polyhedral nanostructure formed by the scaffold routing can be made of single helical domains given an appropriate stapling.

However, not all graphs have all even degree vertices. This is also true for polyhedral graphs and plane triangulations—consider for example the tetrahedral graph of Figure 3.8. From the scaffold routing point of view,

[1]Graphs with self-loops can be considered Eulerian, given that self-loops count twice in the degree.

we have some room to manoeuvre; for instance, the polyhedral beams can be made of two double helices (see Figure 2.5) by the use of more strand base pairs. How can we modify the polyhedral graph, or more generally any graph, so that it becomes Eulerian?

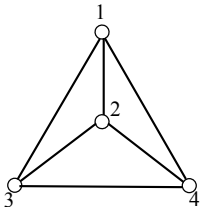
Consider a simple graph G which contains an odd degree vertex v. Can we transform G with a minimal set of operations so that v gets an even degree? Suppose, for the sake of argument, we are allowed to add vertices and edges to G with a restriction: the degree of any other odd degree vertex is not changed, and a new odd degree vertex is not introduced. Is there such an operation that fixes the parity of v? If indeed there is so, we can iteratively fix the odd degree vertices until we get an Eulerian graph. However, such an operation is not possible. By the famous handshaking lemma [60, p.12], there are an even number of odd degree vertices in any graph. Fixing the parity of one odd degree vertex would mean we get a graph with an odd number of odd degree vertices.

On the other hand, we can simultaneously fix two odd degree vertices by only adding edges. Let u, v be two odd degree vertices in a connected graph G and let P be a path from u to v in G. Consider what happens if we add multiedges along the path P. All the internal vertices of P will not change their parity since two edges are added for each of them. Meanwhile, vertices u and v now have one additional edge and thus they have even degrees. For instance, vertices 2 and 6 are of odd degree in the braced grid graph in Figure 4.1(c); two edges are added along the path $(6, 4, 2)$ to make them even as shown in Figure 4.1(d).

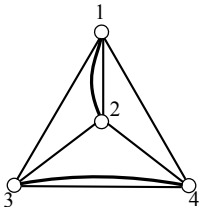
We can now propose a method for fixing the parity of all odd degree vertices of a simple graph. Since we have an even number of odd degree vertices, we can pair up odd degree vertices and fix the pairs one by one. However, pairing and fixing is costly in the number of extra edges added and more edges entail more base pairs in the scaffold. First, we can ensure that there are at most two multiedges between vertices by deleting an even number of extra edges while maintaining the parity of the vertices. For a non-Eulerian polyhedral nanostructure, this ensures that the beams consist of at most two double helices. Second, we can use shortest paths when pairing the edges. For sparse graphs, Johnson's all pair shortest paths algorithm, with a worst-case complexity $\mathcal{O}(|V|^2 \log |V| + |V||E|)$ [17, p.636], is suitable (recall from Theorem 3.1.2 that simple planar graphs are sparse). Nonetheless, we will still have many multiedges unless we pair the vertices efficiently. In fact, for $2k$ odd degree vertices, we have $(2k-1)(2k-3)(2k-5)\cdots 1$ number of possible pairings. To state the third optimization, we restate the problem in a more standard terminology.

A *matching* in a graph G is a set of edges of G no pair of which share an endpoint. A vertex is *covered* by a matching M if it is an endpoint of an edge $e \in M$. A matching M is *perfect* if every vertex is covered. We can then formulate our optimization problem in the following way.

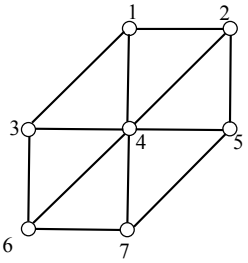
Given a simple graph G with $2k$ odd degree vertices labelled $\{v_1, v_2, \ldots, v_{2k}\}$, we construct a weighted complete graph on $2k$ vertices, K_{2k} labelled with $\{1, 2, \ldots, 2k\}$. The weight w_{e_j} of edge $e \in K_{2k}$ is the length of the shortest path between the endpoints of e in G. The weight of a matching $M = \{e_1, e_2, \ldots, e_l\}$ in K_{2k} is the sum of the weights w_{e_j}. It is easy to see that any k independent edges in K_{2k} form a perfect matching. Our objective then becomes finding the minimum weight perfect matching M^* in K_{2k}. In our software package BScOR, we used Blossom V—an $\mathcal{O}(|V|^3|E|)$ implementation programmed by Kolmogorov [39]—for the minimum weight matching problem. Finally, we add multiedges along the shortest path P from v_i to v_j in G for each edge $e = \{i, j\} \in M^*$.



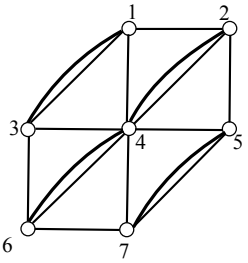
(a) A tetrahedron.



(b) A tetrahedron with multiedges.



(c) A braced grid graph.



(d) A braced grid graph with multiedges.

Figure 4.1: Edge addition to make graphs Eulerian. Added edges are shown in bold lines.

The procedure presented above is a simplified version of an algorithm for the Chinese Postman Problem described by Edmonds and Johnson [24]. In the Chinese Postman Problem, the goal is to find a minimal weight closed walk on a weighted graph which visits every edge at least once. Such a walk is known as a *postman tour*. In our case, all the edge weights on G are

one; thus, the closed walk of minimal weight is the one which has minimal length. Let us denote by $\hat{G}$ the graph formed from G by the edge additions described in the preceding paragraphs; let $\hat{E}$ be the set of added edges. By construction, $\hat{G}$ is Eulerian and thus admits an Eulerian trail T. If we replace each $\hat{e} = \{u, v\} \in \hat{E}$ in T by their parallel edge $e = \{u, v\} \in E(G)$, we get a closed walk in G which visits every edge with the least repetitions (that is, the postman tour in G). Nevertheless, in scaffold routing, the strand cannot physically retrace along an edge. Hence, we use the multiedge representation rather than the multiple edge visit representation when we make a non-Eulerian graph Eulerian. Nevertheless, we will call our procedure the *Chinese postman procedure*. The Chinese postman procedure is implemented as three executables in the BScOR pipeline (cf. Figure 7.1 in the Appendix).

Suppose we transform a non-Eulerian polyhedral graph G to an Eulerian graph $\hat{G}$ by the Chinese postman procedure. Clearly, $\hat{G}$ is still a 3-connected planar graph. However, we will not have a unique embedding in the sense defined in Section 3.3. This is because $\hat{G}$ is not a simple graph. For instance, the newly added multiedge $\{1, 2\}$ of the tetrahedron (drawn in bold in Figure 4.1(b)), can also be drawn on the right-hand side of the original edge. If it were drawn on the right, the embedding obtained would be non-isomorphic to the one shown. Since G has a unique embedding, it is only multiedges in $\hat{G}$ that have a freedom to change their position in the order around their incident vertex. However, as it can be seen in Figure 2.5, two parallel edges need to be consecutive in the orders around their endpoints to form a single beam by stapling. Nevertheless, parallel edges in $\hat{G}$ are always consecutive in the orders around their endpoints.

Proposition 4.1.2. *Let $\hat{G}$ be an Eulerian 3-connected planar graph obtained from a non-Eulerian polyhedral graph G by the Chinese postman procedure. Then, any two parallel edges in $\hat{G}$ are consecutive in the cyclic orders around their endpoints in any planar embedding of $\hat{G}$.*

Proof. Suppose there are two parallel edges e_1 and e_2 that are not consecutive around a vertex u in a planar embedding of $\hat{G}$. Let v be the other endpoint of e_1 and e_2. Since there can be at most two edges between u and v, there must be two other edges e_3 and e_4 incident with u but with endpoints w and x, both different from v, which alternate between e_1 and e_2 in the cyclic order around u. In the embedding on the plane, edges e_1 and e_2 together with u and v form a cycle C which divides the plane into two—its interior and exterior. Vertex w is in the interior of C iff vertex x is in the exterior. Thus, removing u and v disconnects w and x, and hence $\hat{G}$ would not be 3-connected. □

We have so far discussed the necessary and sufficient conditions for a graph to be Eulerian as well as the means to make a graph Eulerian in case it is not. We conclude this section by making two further observations on Eulerian graphs.

Proposition 4.1.3. *An Eulerian graph with at least two vertices is 2-edge-connected (or bridgeless).*

Proof. Let C_1 and C_2 be two components that arise when deleting a bridge e in a graph G. Suppose a trail T starts at $v \in C_1$ in G. For T to visit the edges in C_2, it must visit e first. Once T is in C_2, it cannot return to v since there is no other edge between C_1 and C_2. Hence G is not Eulerian. □

In fact, Fleury's algorithm for finding Eulerian trails operates by bridge detection [60]. By Lemma 3.2.2, we can conclude that the dual of an Eulerian plane graph is loopless. Our second observation is more fundamental and relates to the discussion on duality and plane triangulations in Section 3.2. The proof of the proposition can be found in standard graph theory textbooks (e.g. [59, p.239]).

Proposition 4.1.4. *The dual of a bipartite plane graph is an Eulerian plane graph.*

From our discussion in Section 3.2, we know that plane triangulations are the duals of cubic 3-connected plane graphs. A plane triangulation which is Eulerian is called an *Eulerian triangulation*. From Proposition 4.1.4, we deduce that the dual of a bipartite cubic 3-connected plane graph is an Eulerian triangulation.

4.2 A-trails

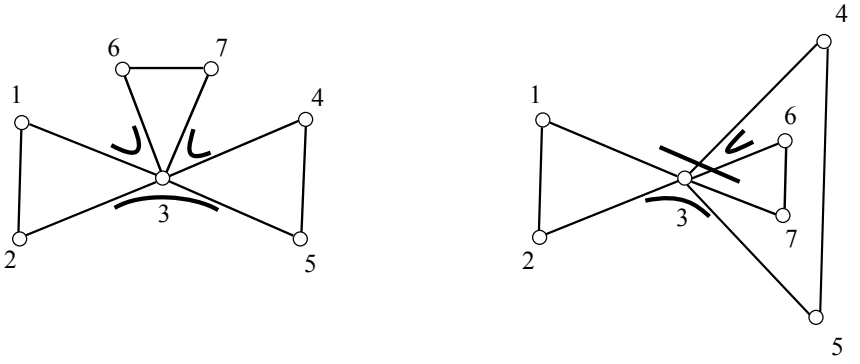
In the design scheme of Högberg's team described in Section 2.3, we mentioned that the scaffold should not cross itself at a vertex. In this section, we describe the desired form of Eulerian trails for such a restriction.

Recall that a plane graph defines a cyclic order of the incident edges to each vertex. We also mentioned that a graph that can be embedded on a plane can also be embedded on a sphere and vice versa. In general, graphs can also be embedded on other surfaces such as the torus [59]. The criteria remain the same: the images of edges cannot cross. If the surface is 'orientable', the embedding defines a cyclic order of the edges [27]. Hence, we can obtain a combinatorial embedding from graphs embedded also on other surfaces.

Given a graph G with a combinatorial embedding (not necessarily on a plane), we say an Eulerian trail T is an *A-trail* if for any segment (e_1, v, e_2) in T, e_1 and e_2 are consecutive in the cyclic order of edges around v. Since a plane graph defines a combinatorial embedding, we can speak of an A-trail for a plane graph. For a 2-connected plane graph, two edges are consecutive in an order around a vertex if and only if they lie on the same face boundary [4]. If a graph has a cut-vertex, this is not necessarily the case; in Figure 4.2(a), edges $\{1,3\}, \{3,4\}$ lie on the boundary of the outer face but are not next to each other in the cyclic order around vertex 3. For a 2-connected plane graph, an A-trail can be defined as a trail in which any two successive edges lie on the same face boundary. As an analogy, a driver that would always turn left or right but never cross an intersection would be following an A-trail; a draft animal which only responds to "Gee" and "Haw" commands likewise. Also observe the scaffold routing in Figure 2.5 and how the scaffold routes at the vertex are compatible with A-trail transitions at the vertex.

It is important to note that a planar graph may admit an A-trail in one embedding but not in another. Figure 4.2 illustrates this idea. The embedding in Figure 4.2(a) has an A-trail $(1,3,6,7,3,4,5,3,2,1)$. On the other hand, the embedding in Figure 4.2(b) does not have any. To see why, assume without loss of generality, the trail starts at vertex 2 and moves to vertex 3. The trail would then either have to turn left to vertex 1 or to the right to vertex 5. If it turns left, it will have to stop after visiting edge $\{1,2\}$. If on the other hand it turns right, it will have to follow the transitions at vertex 3 as indicated by the bold lines. The traversal $7, \{7,3\}, 3, \{3,1\}, 1$ would then be problematic as $\{7,3\}$ and $\{3,1\}$ are not consecutive in the order around vertex 3. Notice how vertex 3 is a cut-vertex. However, if a simple planar graph is 3-connected, it has a unique embedding, up to isomorphism, by Whitney's theorem (page 30). It is easy to see that a planar graph admits an A-trail in an embedding O_1 if and only if it admits an A-trail in an embedding O_2 isomorphic to O_1. As we have noted before, two edges are consecutive in a cyclic order around a vertex if and only if they are consecutive in a reversal of the order of the edges.

There is also a related but different type of Eulerian trail. An Eulerian trail T on a plane graph G is called *non-crossing* if there are no two edge pairs (e_1, e_3) and (e_2, e_4) incident to a vertex v, where e_3 succeeds e_1 and e_2 succeeds e_4 in T, such that e_2 and e_4 alternate between e_1 and e_3 in the cyclic order around v (see Figure 4.3). The trail $(1,3,7,6,3,4,5,3,2,1)$ for the embedding in Figure 4.2(b) is a non-crossing trail: even though edges $\{1,3\}$ and $\{3,7\}$ are not consecutive around vertex 3, neither pair $\{\{6,3\},\{3,4\}\}$ nor $\{\{5,3\},\{3,2\}\}$ alternate between $\{\{1,3\},\{3,7\}\}$ around vertex 3. If one allows a small space in the neighbourhood of a vertex, one can draw a non-



(a) An embedding with an A-trail.

(b) An embedding without an A-trail, but with a non-crossing trail.

Figure 4.2: An A-trail and a non-crossing trail and two plane graphs. The transitions at vertex 3 are shown in bold lines.

crossing trail by continuous movement without any disruptions [58].

Any Eulerian plane graph admits a non-crossing trail; a short proof can be found in Tsai and West [58]. Their proof is constructive and gives rise to an algorithm. The algorithm works by locally readjusting crossings at a vertex. We show here how a crossing at a degree four vertex can be adjusted. Suppose we have an Eulerian trail $T = (\dots, e_1, v, e_3, S, e_4, v, e_2, \dots)$ for an Eulerian plane graph G with a crossing at a vertex v whose incident edges, in the cyclic order, are e_1, e_2, e_3, e_4. The trail $T' = (\dots, e_1, v, e_4, S^R, e_3, v, e_2, \dots)$, where S^R is S reversed, is a trail with no crossing at v. The local adjustment is illustrated in Figure 4.3.

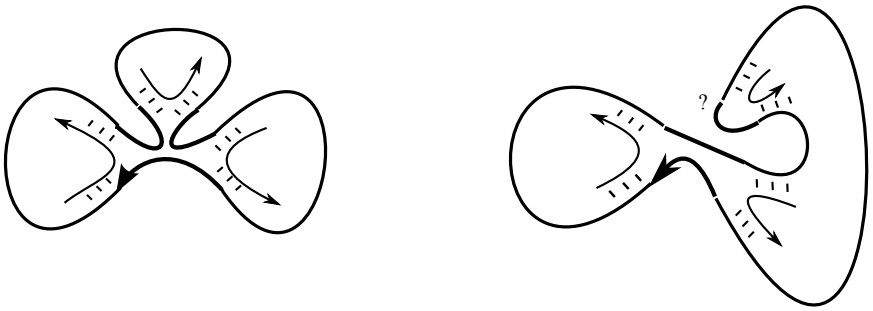
(a) A trail with a crossing at a vertex.

(b) A local adjustment of the trail with no crossing.

Figure 4.3: Local adjustment of a crossing trail at a vertex.

Since there is an insurance than any Eulerian plane graph has a non-crossing trails, we might ask if we can staple a scaffold that has been routed according to a non-crossing trail. Unfortunately, stapling is not straightfor-

ward for a scaffold routed according to a non-crossing trail. Figure 4.4(a) shows how we can staple a scaffold routed according to an A-trail as in Figure 4.2(a). For a scaffold routed according to a non-crossing trail as shown in Figure 4.4(b) (corresponding to the non-crossing trail in Figure 4.2(b)), one stapled segment (indicated by the question mark in Figure 4.4(b)) remains free as it will not form a vertex.



(a) Stapling a scaffold routed according to an A-trail.

(b) An incomplete stapling in a non-crossing trail.

Figure 4.4: Eulerian trails and scaffold stapling. Bold lines are scaffolds; thin lines are staples. Arrows indicate the $5'$ to $3'$ direction.

Thus, for the purpose of scaffold routing, we have to find A-trails. The concept of an A-trail and a non-crossing Eulerian trail coincide if the maximum degree of a plane graph is four. Indeed, a non-crossing trail on such a graph would also be an A-trail. Since a 3-connected graph cannot have a degree two vertex, any Eulerian polyhedral graph with a maximum degree four is, in fact, 4-regular. If the target DNA beam-frame nanostructure is Eulerian but does not contain a degree six or more vertex, we can efficiently find an A-trail by simply finding a non-crossing trail and thus a suitable scaffold routing path for proper stapling. If there is at least one vertex of degree six or more as in Figure 4.2(b), there is no always applicable local adjustment of an ordinary Eulerian trail for obtaining an A-trail. Otherwise, we would be able to find an A-trail for the plane graph in Figure 4.2(a); but, we have argued that the plane graph does not admit an A-trail. How then can we efficiently find an A-trail or even decide if an Eulerian plane graph has one? Given an Eulerian trail of a plane graph, it is easy to check whether it is an A-trail; thus, the problem is in NP, so the question is whether it is an NP-hard problem.

4.3 Complexity considerations

Bent and Manber [9] studied the problem of finding A-trails[2] in the context of finding an optimal torching path for a stock of metal sheets. In their paper they showed, by reduction from SAT, that deciding whether an Eulerian graph with a combinatorial embedding has an A-trail is NP-complete. Their reduction yields an embedding on a plane and thus deciding whether a plane graph has an A-trail is also NP-complete. However, their reduction does not necessarily result in polyhedral graphs.

As discussed in Section 3.3, we are concerned with scaffold routing paths for polyhedral graphs. We have noted that even though deciding whether an Eulerian planar graph has an A-trail depends on the embedding, polyhedral graphs have a unique embedding up to isomorphism. As such, the question of existence of A-trail becomes independent of the embedding for polyhedral graphs. Nevertheless, Anderson and Fleischner [23] have shown that deciding whether an Eulerian polyhedral graph consisting of only 3-cycles and 4-cycles as face boundaries admits an A-trail is also NP-complete.

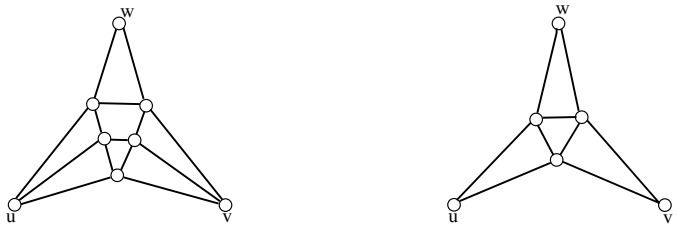
Since the class of Eulerian polyhedral graphs is a superset of those only consisting of 3-cycle and 4-cycle faces, the problem of deciding whether an Eulerian polyhedral graph admits an A-trail is also NP-complete. In fact, the class of graphs under our consideration are Eulerian 3-connected planar graphs (which need not be simple) as argued in Section 4.1. The class of Eulerian 3-connected planar graphs subsumes Eulerian polyhedral graphs; thus we cannot efficiently decide whether an Eulerian 3-connected planar graph admits an A-trail, unless $\mathsf{P} = \mathsf{NP}$.

Anderson and Fleischner [23] proved the NP-completeness result for Eulerian polyhedral graphs by a reduction from the Hamiltonian cycle problem on cubic polyhedral graphs. The cube and tetrahedral graphs in Figure 3.8 are examples of cubic polyhedral graphs. For a cubic polyhedral graph G, they first form its dual G^*. By Theorem 3.2.4, the dual G^* is a plane triangulation. However, G^* need not be Eulerian (e.g. the tetrahedral graph is self-dual). In order to make G^* Eulerian, they insert the graph H_1 (where u, v, w are vertices of G^*) shown in Figure 4.5 into a selected set of faces of G^*. For each edge that gets doubled (or retraced) in a postman tour of G^*, one of the two faces which the edge bounds is selected arbitrarily. H_1 is then inserted into each selected face. Note that since G^* was a plane triangulation, only one of the edges bounding a face can get doubled in its postman tour. Hence, the selected faces into which the H_1's are inserted are distinct.

[2]The authors use the phrase 'non-intersecting Eulerian trails' to what we refer to as A-trails in this thesis.

Finally, in all the faces of G^* into which H_1 was not added, the graph H_0 is inserted.

For instance, if G is the tetrahedral graph, then G^* is also a tetrahedral graph. Suppose the plane graph in Figure 4.1(a) is a representation of G^*: if H_1 is inserted into faces $(1, 3, 2)$ and $(3, 4, 2)$, then H_0 would be inserted into the face $(1, 2, 4)$ and the unbounded face $(1, 3, 4)$.



(a) The graph H_1. (b) The graph H_0.

Figure 4.5: Graphs added in NP-completeness proof by Anderson and Fleischner. For the graph H_1, u and v are endpoints of the edge $\{u, v\}$ that gets retraced in the postman tour of G^*. The vertices u, v and w are on the boundary of the face into which the graphs are inserted.

If G is bipartite, then G^* is Eulerian by Proposition 4.1.4. Hence, graph H_1 is not added into any of the faces. All faces of G^* will contain inserted copies of H_0. Even after the addition of H_0, G^* is an Eulerian triangulation. If we wanted to show that the decision problem on the existence of A-trails restricted to Eulerian triangulations remains NP-complete, we would only need to show that the Hamiltonian cycle problem remains NP-complete for bipartite, cubic, polyhedral graphs (we can use the same reduction with only H_0's inserted). The Hamiltonian problem on bipartite, cubic, polyhedral graphs has a long and interesting history.

In 1880, Tait conjectured that all cubic polyhedral graphs are Hamiltonian, showing that if the conjecture is true, the four colour theorem[3] on planar graphs follows [27]. Tutte, in 1946, found a counterexample to Tait's conjecture with 46 vertices. Counter-examples with fewer vertices have since been found. Tutte, modified Tait's conjecture by replacing planarity with bipartiteness, claiming that every bipartite cubic 3-connected graph is Hamiltonian. Again, Tutte's conjecture was proven false by Horton who first gave a counter-example on 96 vertices. Barnette combined Tait's and Tutte's conjectures, claiming that every bipartite cubic polyhedral graph is Hamiltonian. Barnette's conjecture remains open to this day [30].

[3]The four colour theorem states any planar graph can be 4-vertex-coloured [59, p.260].

Fleischner [27] further develops a complete equivalence between A-trails on Eulerian triangulations and Barnette's conjecture. Hence, we have the following conjecture by Fleischner.

Conjecture 4.3.1 (Fleischner)**.** Every Eulerian triangulation has an A-trail.

What does Fleischner's conjecture imply in the case scaffold routing on rigid polyhedral beam-frameworks? If a rigid polyhedral framework has an Eulerian skeleton (as is the case for instance with the octahedron), then its projection is an Eulerian triangulation. If Fleischner's conjecture is true, then there is a scaffold routing path so that the scaffold can be stapled appropriately. Since the conjecture remains open, we do not know if there are any Eulerian triangulations which do not admit A-trails, nor do we know if the problem of deciding whether such graphs admit A-trails is NP-complete.

We remark here that the difficulty of finding an Eulerian triangulation which does not have an A-trail is not because of the triangularity of the faces, but also because of connectivity of plane triangulations (we noted that plane triangulations with at least four vertices are 3-connected). To illustrate this, we consider the graph shown in Figure 4.6. The Eulerian plane graph shown in the figure is a near-triangulation (all its faces other than the unbounded face are 3-cycles). It is easy to check that the plane graph does not have an A-trail. Note that the graph has multiple cut-vertices. In fact, Fleischner [27] further conjectures that every Eulerian 4-connected planar graph has an A-trail.

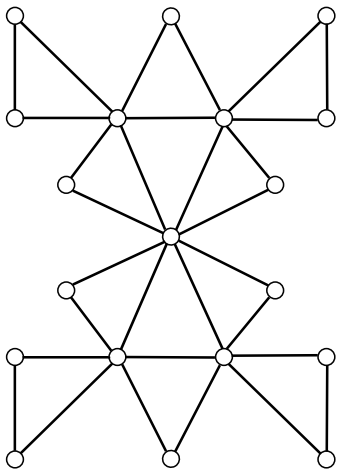


Figure 4.6: A near-triangulation without an A-trail.

Given the NP-completeness result in general, we cannot expect to develop a polynomial time algorithm for finding A-trails in Eulerian 3-connected plane graphs. In Chapter 5, we describe a backtracking search algorithm for finding A-trails in arbitrary Eulerian plane graphs. Nevertheless, we tune the algorithm to perform best on Eulerian triangulations.

Chapter 5

A backtracking algorithm for A-trails

In Chapter 4, we restated the problem of scaffold routing on polyhedral beamframes in the graph theoretic terms of Eulerian trails. In Section 4.2, we showed that the problem of scaffold routing for polyhedral frameworks coincides with the problem of finding A-trails in the polyhedral graphs. In Section 4.1, we described the Chinese postman procedure for making a non-Eulerian graph Eulerian. When transforming a non-Eulerian polyhedral graph by the Chinese postman procedure, we obtain an Eulerian 3-connected planar graph which has multiedges. In Section 4.3, we argued that the problem of finding A-trails in an Eulerian 3-connected plane graph is NP-complete.

In this chapter, we present a backtracking algorithm for finding A-trails in Eulerian plane graphs. First, we structure the search based on an observation on A-trail transitions at a vertex. Second, we present a result which allows the search to backtrack without necessarily reaching leaf nodes of the search tree. Finally, we introduce a simple enumeration heuristic for better pruning of the search tree. We then study the performance of the enumeration heuristic by some run time experiments on three graph families. The A-trail search algorithm is the last component in the pipeline of our software package BScOR (cf. Figure 7.1 in the Appendix).

5.1 Vertex parities

Consider again the embedding in Figure 4.2(b). We argued that the two possible extensions of a trail starting from vertex 2 and moving along to vertex 3 would not yield an A-trail. A naive backtracking scheme would attempt to extend subtrails with the possible extensions (at each step turning

left or right) and check for a trail which visits every edge. If an Eulerian plane graph has a minimum degree four—as is the case in an Eulerian triangulation with at least four vertices—this entails a search space of approximate size $2^{|E|}$.

We first claim that instead of making a choice on two possible successors for each edge, we only need to make a choice on two sets of transitions at each vertex for a plane graph. In such a search, after a successor of an edge has been selected at a vertex, all the other transitions at the vertex will be set. This would reduce the search space to an approximate size of $2^{|V|}$, which is equivalent to $2^{|E|/3+2}$ in an Eulerian triangulation. We now state our observation in Lemma 5.1.1 (also Lemma VI.53 in [27]).

Lemma 5.1.1. *Consider a vertex v in an Eulerian plane graph G with degree d and incident edges in counter-clockwise cyclic order $(e_1, e_2, \dots, e_d)$. Suppose an A-trail T visits e_1 oriented towards v. Then, either*
i) $T = (\dots, w, e_1, v, e_2, \dots, e_3, v, e_4, \dots, e_{d-1}, v, e_d, \dots)$ or
ii) $T = (\dots, w, e_1, v, e_d, \dots, e_{d-1}, v, e_{d-2}, \dots, e_3, v, e_2, \dots)$

Proof. If $d = 2$, then the claim is trivially true. Thus, suppose $d \geq 4$. By definition of an A-trail, T must either choose e_2 or e_d. Suppose T visits e_2 as in case (i); then e_2 will be oriented outwards from vertex v. We claim that the next edge incident to v to be visited by the trail will be e_3.

Suppose it is not; that is, there is an edge $e_j, j > 3$ which is visited next, such that $T = (\dots, w, e_1, v, e_2, x, \dots, y, e_j, v, \dots, e_3, \dots)$. Then, $(x, \dots, y)$ is a walk and thus contains a path P from x to y [59, p.21]. Since e_j was the next incident edge of v to be visited by the trail, P does not contain v. Then, $C = (v, e_2, P, e_j, v)$ is a cycle . The cycle C divides the plane to its interior and exterior. Edge e_3 is in the interior of the cycle if and only if e_1 is outside of it. If T visits an edge in the same region as e_3 after visiting e_j, it cannot go back to w without crossing the cycle. On the other hand, if T visits an edge in the same region as e_1, then it cannot visit e_3 without crossing the cycle. The problem is illustrated in Figure 5.1. In the figure, e_3 is in the interior of the cycle and e_1 is in the exterior. Hence, if T visits any edge $e_j, j > 3$ before e_3, it cannot visit e_3 and would not be an Eulerian trail. Thus, e_3 is the next visited edge. It is oriented towards v, and e_4 oriented outwards will directly follow since e_2 has already been visited.

If $d = 4$, we would be done. If $d \geq 6$, repeat the above argument taking e_5 as e_3 and setting $j > 5$. We conclude that if T visits e_2 in the first step, then it must visit the edges according to case (i). Analogous reasoning shows that if T first chooses edge e_d, it must visit the edges according to case (ii). □

Lemma 5.1.1 states that an A-trail visits the edges of a vertex so that

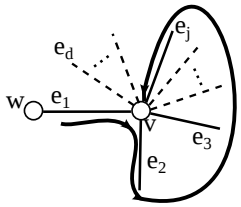
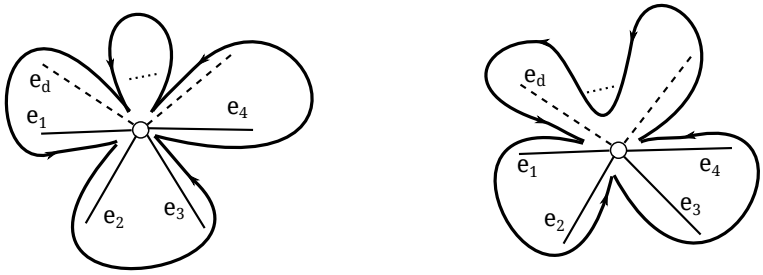


Figure 5.1: Visiting e_j, $j > 3$ is prohibited after visiting e_2.

either even index edges follow odd ones as in case (i) or odd ones follow even index edges as in case (ii) with the orientation reversed. Here, it is assumed that the order has been defined so that e_1 is the first indexed edge. Let us say a vertex v has an *even parity* with respect to an A-trail if the trail makes the transitions in v as in case (i) (or its reverse). Similarly, we say a vertex has an *odd parity* with respect to an A-trail if the trail makes the transitions at the vertex as in case (ii) (or its reverse). The two possible parities are shown in Figure 5.2. An alternative interpretation is that the edges are traversed according to the counter-clockwise order as in case (i) or clockwise order as in case (ii) [27].



(a) Even parity. (b) Odd parity.

Figure 5.2: Vertex parities.

An A-trail thus partitions the vertices V, of an Eulerian plane graph into two groups: those which have an even parity with respect to the trail, V_e, and those which have an odd parity with respect to the trail, V_o. What can we infer about the possibility of obtaining an A-trail from a certain partition of V? To guide us in answering this question, we introduce the notion of a *splitting* at a vertex.

Let us take the case of scaffold routing as an example. Observe how vertices are formed by stapling. If we ignored the staples, we would have

the scaffold routed up to a vertex but never actually reaching the point corresponding to the vertex. The situation is exemplified by the setting on in Figure 4.4(a). Imagine a new set of vertices are created at each point the routed scaffold turns at a given vertex. The newly created vertices would constitute the split of the given vertex.

To state notion of a splitting formally, consider a vertex v in an Eulerian plane graph G with a degree $d \geq 4$, and with incident edges in counter-clockwise order $e_1, e_2, \ldots, e_d$. An *even-splitting* of v in G is a replacement of v by $d/2$ degree two vertices $v_1, v_2, \ldots, v_{d/2}$ such that e_{2i-1}, e_{2i} become incident to v_i, for $1 \leq i \leq d/2$. Similarly, an *odd-splitting* of v in G is a replacement of v by $d/2$ degree two vertices $v_1, v_2, \ldots, v_{d/2}$ such that e_{2i-2}, e_{2i-1} become incident to v_i, for $1 \leq i \leq d/2$, with $e_0 \equiv e_d$. We consider that the identities of edges remain unchanged after splitting, it is only that they attain new endpoints.

When all degree four or more vertices of a graph are split, the graph becomes 2-regular but potentially disconnected. Figure 5.3 shows the two possible splittings of the plane graph in Figure 4.2(a) at vertex 3. We fix the counter-clockwise order at vertex 3 as $(\{2,3\}, \{5,3\}, \{4,3\}, \ldots, \{1,3\})$. An even-splitting at vertex 3 gives the connected cycle shown on the left while an odd-splitting results in the disconnected graph on the right. The even-splitting corresponds to the A-trail $(1, 3, 6, 7, 3, 4, 5, 3, 2, 1)$.

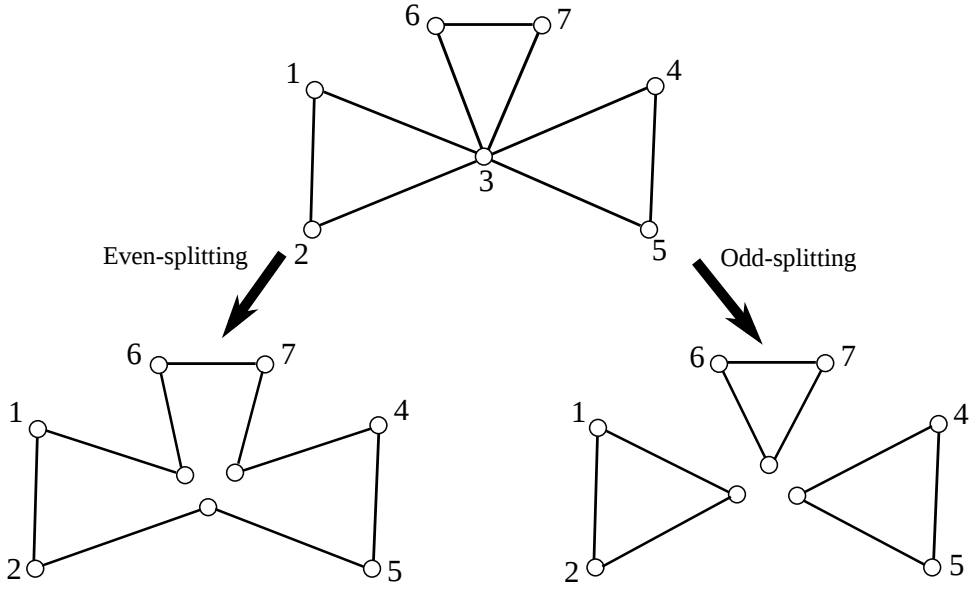


Figure 5.3: The two possible splittings of a vertex.

We can now characterize Eulerian plane graphs which have A-trails.

Theorem 5.1.2. *Let G be an Eulerian plane graph with a vertex set V. Then, G has an A-trail if and only if there exists a partition $\{V_e, V_o\}$ of V such that an even-splitting of vertices in V_e and an odd-splitting of vertices in V_o keeps the graph connected.*

Proof. (only if) Suppose G has an A-trail T. Let V_e be the set of vertices with an even parity and let V_o be the set of vertices with odd parity with respect to T. Let $\hat{G}$ be the graph obtained by an even-splitting of the vertices in V_e and an odd-splitting of the vertices in V_o. Let $\hat{T}$ be the trail with the same sequence of edges as in T but with the vertices replaced by the split vertices in $\hat{G}$. Every vertex in $\hat{G}$ appears in the trail $\hat{T}$, because its two incident edges appear in concession in T. Indeed, if a vertex $\hat{v}_i$ in $\hat{G}$ is the ith split vertex of an evenly split vertex v in G with the incident edges e_{2i-1}, e_{2i} in the counter-clockwise cyclic order around v, then either (e_{2i-1}, v, e_{2i}) or (e_{2i}, v, e_{2i-1}) is a segment of T since v has an even parity with respect to T. Hence, either (e_{2i-1}, v_i, e_{2i}) or (e_{2i}, v_i, e_{2i-1}) is a segment of $\hat{T}$. Likewise, if a vertex $\hat{w}_i$ in $\hat{G}$ is the ith split vertex of an oddly split vertex w in G with the incident edges e_{2i-2}, e_{2i-1} in the counter-clockwise cyclic order around w, then either (e_{2i-2}, w, e_{2i-1}) or (e_{2i-1}, w, e_{2i-2}) is a segment of T. Hence, either $(e_{2i-2}, w_i, e_{2i-1})$ or $(e_{2i-1}, w_i, e_{2i-2})$ is a segment of $\hat{T}$. Thus, there is a path between any two vertices in $\hat{G}$ since there is a sequence of edges in $\hat{T}$ between the vertices.

(if) Suppose there is a partitioning $\{V_e, V_o\}$ of G such that an even-splitting of V_e and an odd-splitting of V_o leaves the graph connected. The graph obtained after the splitting is 2-regular and connected, and thus a cycle. The sequence of edges in the cycle is an A-trail in G. □

By Theorem 5.1.2, we can check for the existence of an A-trail by checking the connectivity of splittings of all possible partitions of V. We conveniently assume that a splitting of a degree two vertex will leave it untouched. We will next formulate the algorithm more precisely and study its complexity.

5.2 A splitting schedule heuristic

We do not need to split all the vertices at once for each possible partition. Indeed, if we know a certain split at a vertex results in a disconnected graph, any further splittings of other vertices will not make the graph connected. Formally, suppose that we set-up a *splitting schedule* for the vertices. That is, we enumerate the vertices in V as $(b_1, b_2, \ldots, b_{|V|})$. Now, suppose we have

split the vertices from b_1 up to b_i so that some subset undergo an even-splitting and the rest undergo an odd-splitting. Further assume that the graph remains connected up to the splitting of b_i. If an even-splitting of b_{i+1} disconnects the graph, no sequence of splittings of b_j for $j > i+1$ will yield a partition giving rise to an A-trail. Thus, we do not need to extend the partial solution (i.e. the splittings of vertices b_1 to b_i under consideration) and thus we should backtrack to make an odd-splitting of b_{i+1}.

Recall that, if the maximum degree in an Eulerian plane graph is four, we necessarily have an A-trail since a non-crossing trail is also an A-trail in such a graph. Hence, there exists a splitting of the vertex set of the graph induced by the non-crossing trail that keeps the graph connected. Now, given an Eulerian plane graph which potentially has vertices of degree greater than or equal to six, set-up a splitting schedule of the vertices of the graph so that all degree four vertices are split last. If the graph remains connected up to the splitting of all degree six or more vertices, we know that there exists an A-trail without splitting the degree four vertices. Thus, we can ignore the splitting of degree four vertices and only schedule vertices of degree six or more. We can thus refine our search to branch on vertices of degree six or more. Since our search algorithm only branches on vertices with degree six or more, we call such vertices *branch vertices*. We now present the refined backtrack search algorithm, Algorithm 1, for deciding whether an Eulerian plane graph has an A-trail and for finding an A-trail if there exists one.

If the graph has multiedges, these edges can be subdivided by additional vertices to obtain a simple plane graph without affecting the outcome of the algorithm. Subdividing of multiedges at most doubles the number of edges, and increases the number of vertices by the number of multiedges. If the input graph has at most one multiedge per each pair of vertices, as it is the case for a graph obtained from a simple graph by the Chinese postman procedure (cf. Section 4.1), the number of vertices increases by at most the number of edges in the 'underlying' simple graph. Since a simple planar graph has at most $3|V| - 6$ edges by Theorem 3.1.2, the number of additional vertices is a constant multiple of the original number of vertices if the graph is obtained from a simple graph by the Chinese postman procedure. More importantly, the number of branch vertices does not increase by the subdivision of multiedges of any graph (the complexity of the algorithm primarily depends on the number of branch vertices as discussed later). Thus, we can assume that the input is a simple plane graph. A simple plane graph can be represented as an adjacency list where the adjacent vertices of a given vertex are listed according to the cyclic order of their incident edges around the given vertex.

The algorithm first enumerates the branch vertices as indicated in line 0.2 in Algorithm 1. If there are no branch vertices, the algorithm resorts to

Input : An Eulerian plane graph $G = (V, E)$
Output: An A-trail T on G if there exists one, no otherwise

```
0.1  has_trail ← true;
0.2  Enumerate the branch vertices in V as B ← (b_1, b_2, ..., b_k);
0.3  if B is not empty then
0.4  |  if splitAndCheck(G, B, 1, odd) returns false then
0.5  |  |  has_trail ← false;
0.6  |  |  if splitAndCheck(G, B, 1, even) returns true then
0.7  |  |  |  has_trail ← true;
0.8  |  |  end
0.9  |  end
0.10 end
0.11 if has_trail is true then
0.12 |  T ← an Eulerian trail of G;
0.13 |  T ← T with local adjustment on degree four vertices;
0.14 |  T ← T where each instance of a split vertex is substituted by
     |  its branch vertex;
0.15 |  return T;
0.16 end
0.17 return no;
```

Algorithm 1: A-trail backtrack search.

finding an ordinary Eulerian trail in line 0.12, for instance by Hierholzer's algorithm [37]. If there are any crossings on degree four vertices, the crossings are fixed vertex by vertex as described in Section 4.2 (recall Figure 4.3).

If there are some branch vertices, the algorithm starts the splitting of these vertices with an odd-splitting of the first branch vertex (line 0.4 in Algorithm 1). It then recursively splits succeeding branch vertices according to the *splitAndCheck* Procedure. If at the end of the recursion, the odd-splitting of b_1 succeeds (i.e. the recursion returns true in line 0.4 in Algorithm 1), the algorithm moves on to finding the A-trail corresponding to the successful split (line 0.12 in Algorithm 1). If on the other hand, all possible extensions of an odd-splitting of b_1 result in a failure and the recursion returns false, the algorithm carries out an even-splitting of b_1 as shown in line 0.6. If the even-splitting also fails, then b_1 would neither be in V_o nor in V_e. Thus, the algorithm returns no in line 0.17.

The *splitAndCheck* procedure carries out the splittings in a depth-first manner with a preference to odd-splittings. When the procedure detects that the graph is disconnected, it backtracks—removing split vertices and

```
1.1  splitAndCheck(G, B, i, parity) begin
1.2  |  Split B(i) according to parity;
1.3  |  if G becomes disconnected then
1.4  |  |  Patch the edges back to B(i);
1.5  |  |  Remove the split vertices;
1.6  |  |  return false;
1.7  |  else
1.8  |  |  if i < B.size then
1.9  |  |  |  if splitAndCheck(G, B, i + 1, odd) returns true then
1.10 |  |  |  |  return true;
1.11 |  |  |  else
1.12 |  |  |  |  if splitAndCheck(G, B, i + 1, even) returns true
     |  |  |  |  then
1.13 |  |  |  |  |  return true;
1.14 |  |  |  |  else
1.15 |  |  |  |  |  Patch the edges back to B(i);
1.16 |  |  |  |  |  Remove the split vertices;
1.17 |  |  |  |  |  return false;
1.18 |  |  |  |  end
1.19 |  |  |  end
1.20 |  |  else
1.21 |  |  |  return true;
1.22 |  |  end
1.23 |  end
1.24 end
```

Procedure `splitAndCheck`.

patching edges to get back to the original vertices (lines 1.4 to 1.6). If both splittings of a branch vertex b_{i+1} result in failure (lines 1.9 and 1.12), the procedure further backtracks (line 1.15), patching the edges to b_i and removing the split vertices of b_i. Once the recursion has reached the stage where $i = B.size$, the procedure then splits the last the last branch vertex in the splitting schedule. If the graph remains connected after the last branch vertex is split, the recursion ends with a positive result (line 1.21).

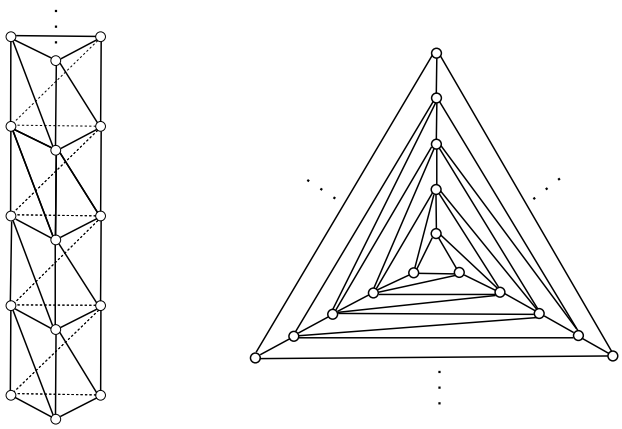
If the recursion ends with a positive result, the algorithm returns to the main routine to find an Eulerian trail on the graph with split branch vertices (line 0.12 in Algorithm 1). The split graph has a maximum degree of four and hence an A-trail. Since the trail comprises original degree two and four vertices and split vertices of the branch vertices, the algorithm substitutes the split vertices by their corresponding branch vertices at line 0.14. The required bookkeeping can be maintained by a map data structure between split vertices and their corresponding branch vertices.

The correctness of the algorithm is evident from the discussion preceding the presentation of the algorithm. If the algorithm outputs an A-trail, then the recursion must have ended with a positive output. The recursion can only end with a positive output if there is a sequence of splittings up to the last branch vertex such that the graph remains connected. Let V_o be the set of vertices which underwent an odd-splitting and let V_e be the set of vertices which underwent an even-splitting in such a sequence. Since the graph remains connected, the graph has an A-trail by Theorem 5.1.2. On the other hand, if an Eulerian plane graph has an A-trail, then there is a partition of its vertices into V_o and V_e such that an odd-splitting of the vertices in V_o and an even-splitting of the vertices in V_e which keeps the graph connected. When the algorithm splits vertices according to this partition, the sequence of recursive calls corresponding to the splittings will not backtrack as the graph remains connected. Thus, the recursion will return true and the calling function will find the A-trail corresponding to the partition.

Let us now consider the complexity of the algorithm. Hierholzer's algorithm is known to take linear time [37, p.48]. Further, fixing of crossings at line 0.13 of Algorithm 1, would take $\mathcal{O}(|E|^2)$ in the worst case since it comprises of list section reversals. The complexity of the algorithm is predominantly due to the *splitAndCheck* procedure. Considering the main routine as the root node, the complete search tree has $2^{k+1} - 1$ nodes, where $k = |\{v \in V : deg(v) \geq 6\}|$. In the worst case, we carry out splitting, patching and checking of connectivity in each node of the search tree. Checking connectivity by depth-first search takes linear time. Splitting, at line 1.2, entails addition of vertices and addition and removal of edges which can all be done in $\mathcal{O}(|E| + |V|)$ time [55] (which equals $\mathcal{O}(|V|)$ for simple planar

graphs); with a factor of $d/2$ for a degree d branch vertex. Patching of edges and removal of split vertices can also be done in linear time [55]. The algorithm thus has a worst-case complexity of $\mathcal{O}(2^k\Delta|E|)$, where Δ is the maximum degree in G. The main parameter affecting the complexity is the number of branch vertices which begs the question: how many of the vertices in a given simple planar graph are branch vertices?

A simple planar graph must have at least one vertex with degree less than six [60, p.68]. If all degrees are six or more then the number of edges is lower bounded by $6|V|/2 = 3|V|$ which is greater than $3|V| - 6$. Analogous reasoning shows that the average degree of a simple planar graph cannot be above six [56, p.118]. Thus, if there are many high degree vertices, we expect that there are many degree two and degree four vertices, which would entail a large reduction in the search space. Unfortunately, there is even an infinite family of Eulerian triangulations which have a constant fraction of degree four vertices as shown in Figure 5.4(b). The triangulations are two-dimensional representations of 3D towers (depicted in Figure 5.4(a)) envisioned by Högberg's team [36]. The parameter generating the family of Eulerian triangulations is the number of floors in the 3D tower. Only the vertices on the base of the first floor and the vertices on the ceiling of the top most floor have degree four; all the other vertices have degree six. Even for this graph family, the algorithm can save a factor of 2^6 by only considering degree six vertices.



(a) 3D tower family. (b) 2D representation.

Figure 5.4: Eulerian triangulations with a large number of branch vertices.

We can still look for improvements in the algorithm: in particular in the splitting schedule. Thus, we develop a scheduling heuristic based on

conflicts. First, let us define some terms. A *parity configuration* of a set of branch vertices in an Eulerian plane graph G is a Boolean assignment of each branch vertex in the set to the value *odd* or *even*. A set of branch vertices are said to be in *conflict* with respect to a parity configuration if a splitting based on the configuration disconnects G. Note that a parity configuration on a set of branch vertices imposes no order in the splitting sequence of the branch vertices.

Suppose that three branch vertices (u, v, w) in a graph G are in conflict with respect to a configuration (odd, odd, odd). Further suppose, without loss of generality, that u and v are the first vertices that get split in accordance with the configuration. If w is scheduled next, the algorithm will backtrack after detecting an odd-splitting of w disconnects G and then splits w with an even parity. Thus, the search tree is pruned at depth four from its root (where the root, considered to be the main routine, is at depth zero, and the first vertex is split at depth one). The number of nodes visited in the search tree would be reduced by a size of $2^{k-2}-2$. If, however, w is scheduled late, suppose last, then the algorithm will descend to the leaf of the search tree before realizing that an odd-splitting of w disconnects the graph. There would be no reduction in the number of visited nodes in the search tree from the conflicting parity. In general, if w is split at depth t, where we assume the first branch vertex gets split at depth one and the root is at depth zero, the number of visited nodes is reduced by a size of $2^{k-t+1}-2$.

To ensure a significant reduction in the number of visited nodes in the search tree, the branch vertices should be scheduled so that the vertices which are likely to have conflicts in a certain parity configuration are scheduled as close to each other as possible. Now, consider a 3-cycle face in an Eulerian plane graph with branch vertices u, v, w as boundaries as shown in Figure 5.5. The parity configuration indicated by the bold lines is in conflict. As such, there is always some parity configuration of three branch vertices on the boundary of a triangular face which is in conflict. In an Eulerian triangulation in particular, any two vertices v, w that are ends of two edges which are consecutive in a cyclic order around a vertex u, are adjacent to each other (or equivalently u, v and w form a triangular face).

Taking this into account, we have used a simple modification of breadth first search (BFS), which we call *plane breadth first search*, as our splitting schedule heuristic; it is presented in Algorithm 3. The only difference with ordinary breadth first search is in the manner incident edges of a vertex are checked at line 2.13. In plane breadth first search (plane BFS), we check edges according to the cyclic order where one edge is set as the start of the order. In our particular case, branch vertices are added to the splitting schedule, B, as they are first encountered. Plane breadth first search has a

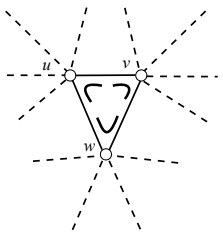


Figure 5.5: A conflicting parity configuration for three branch vertices bounding a triangular face.

worst-case complexity of $\mathcal{O}(|V| + |E|)$.

Input : An Eulerian plane graph $G = (V, E)$
Output: An enumeration, B, of branch vertices ($\{v \in G | deg(v) \geq 6\}$)

```
2.1  B ← empty vector;
2.2  Q ← empty queue;
2.3  for v in V do
2.4  |  color[v] ← white;
2.5  end
2.6  Take any s in G;
2.7  enqueue (Q, s);
2.8  if degree of s ≥ 6 then
2.9  |  Append s to B;
2.10 end
2.11 while Q is not empty do
2.12 |  u ← dequeue (Q);
2.13 |  for edges e in the cyclic order around u do
2.14 |  |  v ← The other endpoint of e;
2.15 |  |  if color[v] is white then
2.16 |  |  |  color[v] ← gray;
2.17 |  |  |  if degree of v ≥ 6 then
2.18 |  |  |  |  Append v to B;
2.19 |  |  |  end
2.20 |  |  |  enqueue (Q,v);
2.21 |  |  end
2.22 |  end
2.23 |  color[u] ← black;
2.24 end
2.25 return B;
```

Algorithm 3: Plane breadth first search.

5.3 Run time experiments

In order to test the improvement we get from the plane BFS enumeration heuristic, we carried out some run time experiments on three graph families. For comparison, we also obtained run time data for a random enumeration (permutation) of the branch vertices. The number of nodes visited in the search tree was taken as the performance metric. Instead of number of vertices, the number of branch vertices was taken as the independent variable. Indeed, two graphs with the same number of vertices may have significantly varying number of branch vertices; yet, the size of the search tree is determined by the number of branch vertices.

We tested the performance of the algorithm in three graph families. The first family is the class of Eulerian triangulations corresponding to the 3D towers depicted in Figure 5.4(a). As noted before, the family is obtained by taking the number of floors as the parameter. For a t floor tower, $t \geq 1$, there are $3t-3$ branch vertices; thus, a t can be obtained for a desired branch vertex count that is a multiple of three. Figure 5.6 shows the performance of plane BFS for branch vertex counts which are the first ten multiples of 99.

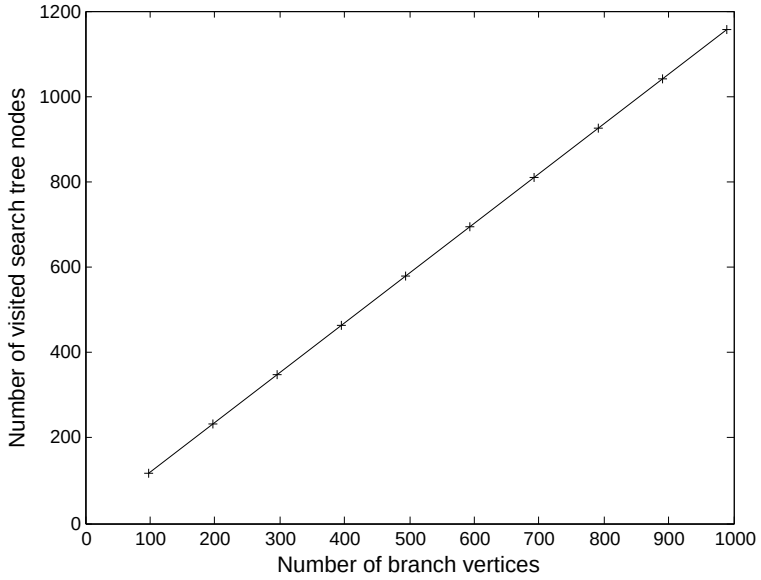


Figure 5.6: Number of visited search tree nodes with plane BFS scheduling for the tower graph family.

There is no randomness in the plane BFS enumeration and the plotted

values are based on single runs of the algorithm. The plot shows a linear trend and even a branch vertex set of size 990 is handled efficiently. All the tested instances had an A-trail.

In comparison, Figure 5.7 shows a box plot on the base-2 log of number of visited search tree nodes based on a random enumeration of the branch vertices. All logarithms shown in the upcoming plots are also to the base 2. Branch vertex set sizes are set starting from 9 to 63 with increments of 9. Each box in Figure 5.7 is based on 21 different random enumerations of the branch vertices.

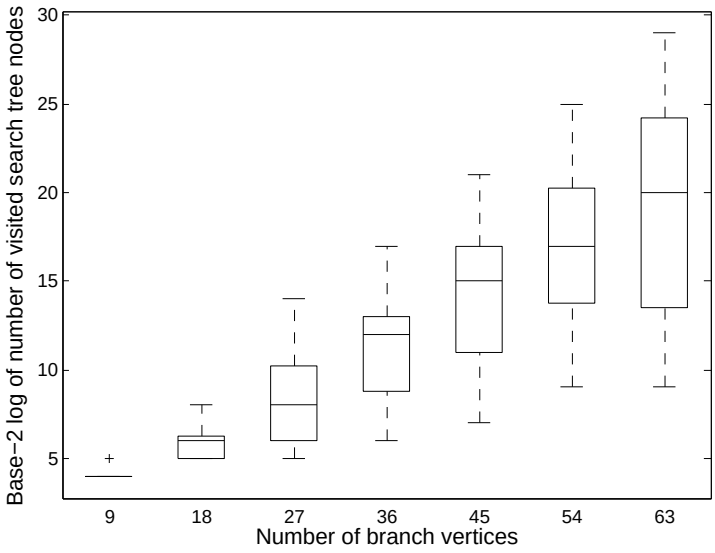


Figure 5.7: The base-2 log of number of visited search tree nodes with random scheduling for the tower graph family.

The mid marks in the boxes indicate the medians. The edges of the box are 25th and 75th percentiles. Points beyond the whiskers are considered outliers. Outliers are indicated by a plus mark and are either larger than $Q3 + 1.5 \times (Q3 - Q1)$ or smaller than $Q1 - 1.5 \times (Q3 - Q1)$, where $Q1$ and $Q3$ are the 25th and 75th percentiles, respectively. This rule also applies to all the box-plots that follow. While the plane BFS heuristic handles branch vertex counts up to 990 efficiently, random enumeration struggles even at the size of 63 where the median is at 2^{20}.

The second family of graphs is based on braced (or triangulated) plane grids, also motivated by our collaboration with Högberg and his team [36]. The bracing makes the structures rigid in the plane [38, p.155]. Two corner vertices are left out to keep the graph 3-connected. The parameters generating the family are the width and height of the grid structure. A 3 by 3

instance of the family is depicted in Figure 5.8. The underlying simple graphs are near-triangulations, but not necessarily Eulerian. Edges can be added to make the graphs Eulerian by the Chinese postman procedure as shown in the figure. All the interior vertices (i.e. vertices not on the boundary of the unbounded face) are branch vertices. To get an approximate count of branch nodes from 200 to 2000 with increments of 200, ten grids of height-by-width dimensions, 11x21, 11x41, 11x61, ..., 11x201, were generated. The result of running the algorithm with plane BFS for this graph family is shown in Figure 5.9.

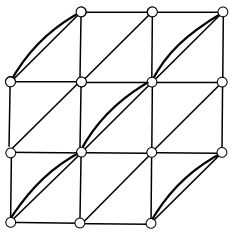


Figure 5.8: The braced grid graph family.

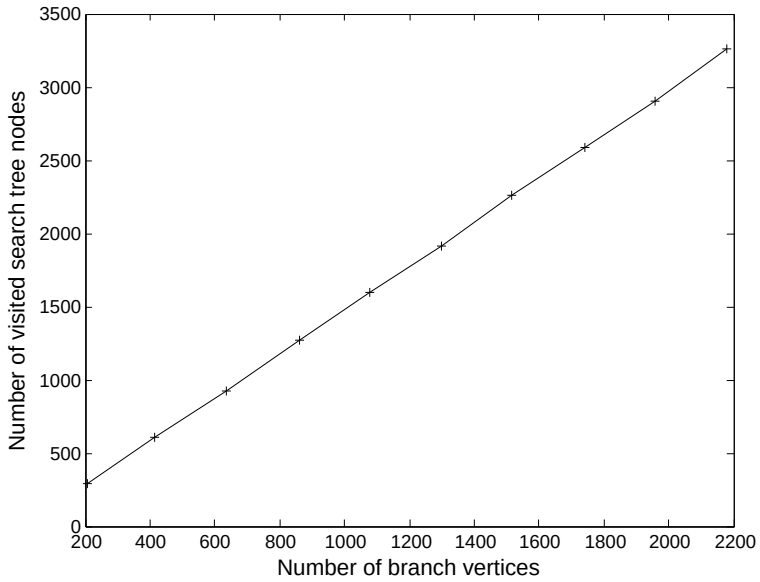


Figure 5.9: Number of visited search tree nodes with plane BFS scheduling for the braced grid graph family.

Once again, the algorithm equipped with the plane BFS heuristic handles the braced grid family efficiently. The number of visited search tree nodes is

only 3260 for a branch node set size of 2178. All the tested plane grid graphs had A-trails.

On the other hand, the algorithm quickly struggles when the branch vertices are scheduled by random enumeration as shown in Figure 5.10. The plane grids corresponding to the branch vertex sizes had dimensions, 6x3, 6x5, 6x7, ..., 6x15. The plotted values in Figure 5.10 are based on 21 random enumerations for each instance size. While the the best random enumeration only visits 2^7 search tree nodes at branch vertex set size of 71, the median random enumeration at the same size visits 2^{13} search tree nodes (more than the amount plane BFS visits at branch set size of 2178).

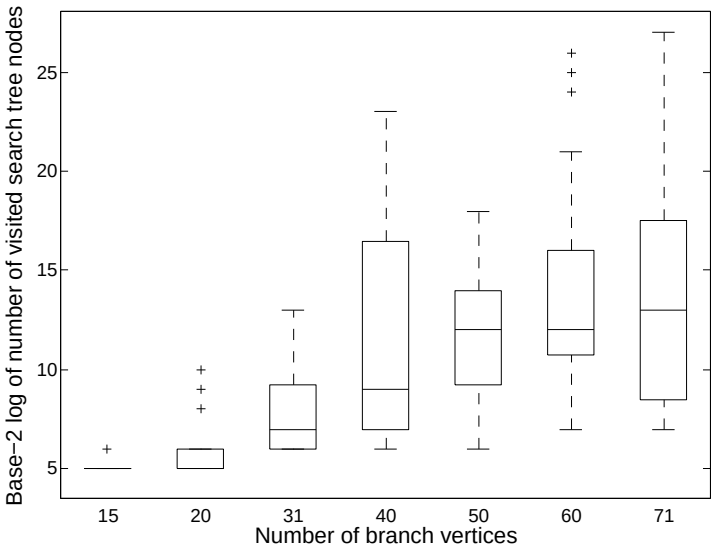


Figure 5.10: The base-2 log of number of visited search tree nodes with random scheduling for the braced grid graph family.

The last family of graphs that were tested are random 2-connected plane graphs. This family was used to investigate the performance of the algorithm on unstructured graphs. These graphs were generated with a random planar graph generating function from the LEDA algorithms library [1]. To generate a graph on n vertices and m edges, a random plane triangulation on $n-1$ vertices is first generated. Then, the last vertex is inserted into a random face of the plane triangulation and connected to the bounding vertices of that face. Edges of the plane triangulation on n vertices are then deleted until m edges are left remaining. In our case, the generated graph is discarded if it is not 2-connected.

Figure 5.11 shows box plots of the base-2 log of number of visited search tree nodes of the algorithm running on plane BFS scheduling of branch ver-

tices. The figure shows the plots for branch vertex sizes up to 100. For each (approximate) instance size, the plane BFS heuristic was tested on 21 random 2-connected planar graphs.

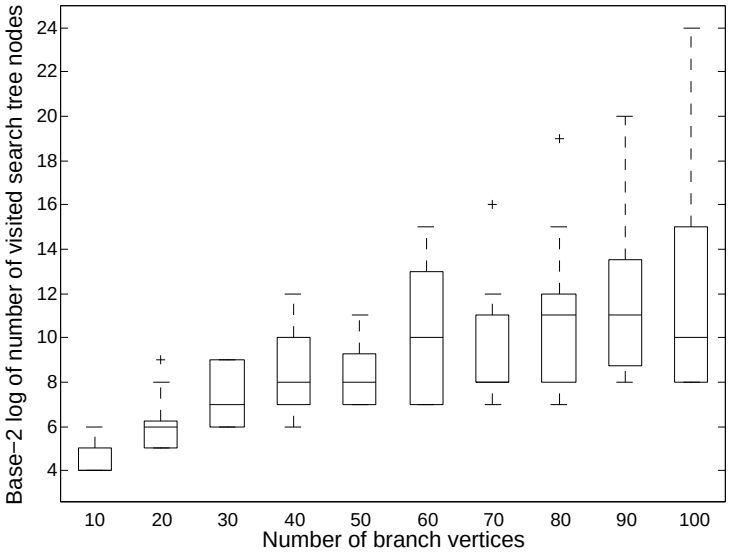


Figure 5.11: The base-2 log of number of visited search nodes with plane BFS scheduling for the random 2-connected plane graphs.

Controlling the exact value of the number of branch vertices was difficult, thus each box has ± 2 in the x-values. The median performance, as indicated by the mid markers in the boxes, was good with a maximum of 2^{11} at sizes 80 and 90. However, the worst-case performance, plotted either as a top whisker or as an outlier, grows rapidly. For branch vertex set size of approximately 100, around 2^{23} nodes in the search tree were visited. This is in stark contrast with the performance of plane BFS on structured graphs. The testing beyond this mark, not shown here, resulted in incomplete data as the algorithm did not halt on most instances within a reasonable time frame. From the set shown in the figure, only two of the tested instances, with sizes 90 and 100, did not admit A-trails.

For testing the random enumeration on random 2-connected plane graphs, the top five best performing instances in the test of plane BFS was selected for each instance size. Eleven runs of the algorithm based on random enumerations were then carried out for each instance. Each box in Figure 5.12 thus represents 55 runs. However, the runs based on random enumeration did not all complete for instances greater than 50. It can be seen from the figure that both the median and the worst-case run time quickly grow.

To summarize the results of our run time experiments, we observed that

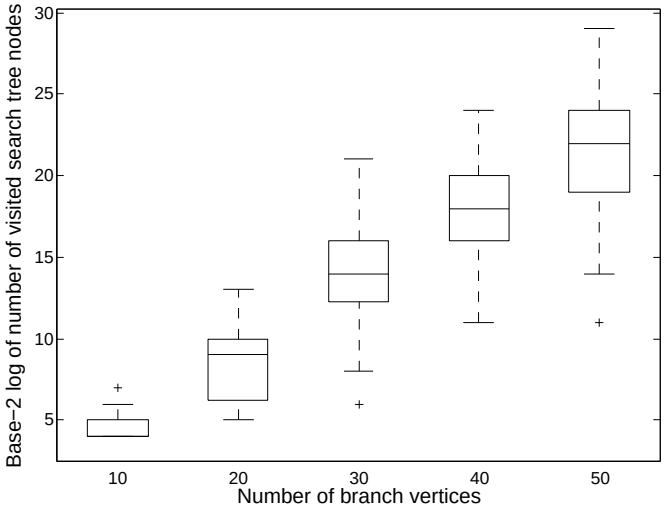


Figure 5.12: The base-2 log of number of visited search tree nodes with random scheduling for the random 2-connected plane graphs.

the plane BFS heuristic considerably outperforms random scheduling on the structured graphs (the tower family and the braced grid family). However, both scheduling methods quickly become intractable for some random instances. Almost all tested instances had an A-trail which lends support to the claim that well connected Eulerian plane graphs are likely to have A-trails.

Chapter 6

Conclusions

In this work, we studied the problem of routing a scaffold on DNA polyhedral beam-frames proposed by Högberg and his team (Section 2.3). We modelled the problem of scaffold routing in graph-theoretic terms and applied graph-theoretic results to tackle the problem. The general setting is that we are given a three-dimensional polyhedral beam-framework as an input, and we wish generate a scaffold routing path as an output if such a path can be found (cf. the software diagram of BScOR in Figure 7.1 in the Appendix).

First, we presented Steinitz theorem (Theorem 3.3.1) which states that polyhedral skeletons are exactly the 3-connected simple planar graphs. Thus, we can ignore the three-dimensional geometric aspects of the framework and study the problem of scaffold routing under an embedding in a plane. Moreover, Whitney's unique embedding theorem (page 30) ensures that we need not worry about the particular embedding we obtain, since a polyhedral graph essentially has a unique embedding. In addition, if the polyhedron is convex, the framework is guaranteed to be rigid if all the faces are triangles (page 28). When the skeleton of a rigid polyhedral framework is projected to a plane, all the faces are triangles. Thus plane triangulations are the projections of rigid polyhedral frameworks and the problem of scaffold routing on rigid polyhedral beam-frameworks can be stated as one of routing in plane triangulations.

Next, we formulated the problem of scaffold routing in terms of Eulerian trails (Chapter 4). We noted that if all the polyhedral beams are to be made of single helical domains, the polyhedral graph needs to be Eulerian. If the graph is not Eulerian, we can add a minimal number of edges by the Chinese postman procedure (page 34) to ensure an Eulerian trail can be found. However, this does not guarantee that the scaffold can be stapled. To ensure that the scaffold can be stapled, we need to find A-trails (Figure 4.4). The problem of deciding whether an Eulerian 3-connected planar graph, the output

of the Chinese postman procedure on polyhedral graphs, admits an A-trail was shown to be NP-complete (Section 4.3). On the other hand, Fleischner's conjecture (Conjecture 4.3.1) states that every Eulerian triangulation admits an A-trail. If the conjecture holds, then there exists a scaffold routing path for any Eulerian rigid polyhedral beam-framework.

We also mentioned Fleischner's conjecture (page 42) that every 4-connected planar graph admits an A-trail. Indeed, in light of the fact that bridges prohibit Eulerian trails (Proposition 4.1.3); cuts prohibit Hamiltonian cycles [59, p.287]; and there are near-triangulations which do not admit A-trails (Figure 4.6); connectivity gives a good platform to study traversal problems of which routing is one kind. Indeed, by Whitney's theorem (Theorem 3.2.1) and its generalization by Menger [59, p.167], well connected graphs have multiple internally disjoint paths between vertices which gives an intuition as why such graphs are easier to traverse. We have also intermittently noted that cuts give flexibility to a framework (Figure 3.7(a)) and flexibility in the embedding (Figure 4.2). Further investigating the relationship between connectivity on the one hand and embedding, routing, and rigidity on the other may be a worthwhile endeavour.

Given the general NP-completeness result, we then introduced a backtracking search algorithm for finding A-trails on Eulerian plane graphs. Moreover, we introduced an enumeration heuristic, plane BFS, to help reduce the number of visited nodes in the search tree. The algorithm, equipped with plane BFS, was shown to significantly reduce the number of visited search tree nodes in well connected planar graphs. Our run time experiments also supported the position that well connected planar graphs generally admit A-trails.

We have reformulated the problem of scaffold routing on polyhedral beam-frameworks as a problem of finding A-trails in their skeletal graphs. Would a scaffold routed according to an A-trail and then stapled using complementary staple strands form the desired nanostructure in a solution?

First, we have concern over the rigidity of the formed nanostructures or more precisely whether our target geometry can be achieved through self-assembly even when the desired structure is a triangular framework. Even though no edge length preserving transformation may take a convex triangular framework to another shape, perhaps another framework with the same set of edge lengths (or congruent faces) may form under self-assembly. For instance, an asymmetric octahedron has a congruent non-convex variant where the shorter apex is pushed in towards the longer one [31]. Interestingly enough, combinatorics has much to say about rigidity [31] and studying its applicability and implications on the rigidity of DNA nanoscale structures may be one possible line of future work.

In addition, we have ignored the geometry of the three-dimensional structure as well as the helical geometry of B-DNA when we studied the problem of scaffold routing. However, the helical geometry of B-DNA dictates that the scaffold should be in a specific phase once it has traversed an edge. If the routing path generated by BScOR is opposite to the natural direction that the scaffold wishes to turn, the helix will come under stress. We are presently investigating the possibility of defining an optimization version of A-trails whereby an unnatural turn for B-DNA under the edge length would be penalized. We are planning to integrate BScOR with vHelix in our continued collaboration with Högberg's team taking the geometric considerations into account.

To conclude, we have used combinatorial techniques to tackle a problem arising from DNA nanotechnology. We hope to give further combinatorial insights to researchers in DNA nanotechnology to help them resolve some of their design challenges.

Bibliography

[1] Algorithmic Solutions Software GmbH. *The LEDA User Manual*, 6.4 ed. Germany, 2008.

[2] Andersen, E. S., Dong, M., Nielsen, M. M., Jahn, K., Lind-Thomsen, A., Mamdouh, W., Gothelf, K. V., Besenbacher, F., and Kjems, J. DNA Origami Design of Dolphin-shaped Structures with Flexible Tails. *ACS Nano 2*, 6 (2008), 1213–1218.

[3] Andersen, E. S., Dong, M., Nielsen, M. M., Jahn, K., Subramani, R., Mamdouh, W., Golas, M. M., Sander, B., Stark, H., Oliveira, C. L., et al. Self-assembly of a Nanoscale DNA box with a Controllable Lid. *Nature 459*, 7243 (2009), 73–76.

[4] Andersen, L. D., Fleischner, H., and Regner, S. Algorithms and Outerplanar Conditions for A-trails in Plane Eulerian Graphs. *Discrete Applied Mathematics 85*, 2 (1998), 99–112.

[5] Areddy, P. K. Computer-Aided Design of Polyhedral DNA Nanostructures. Master's thesis, KTH, 2012.

[6] Argüelles, J. C. The Double Helix Revisited: a Paradox of Science and a Paradigm of Human Behaviour. *Asclepio 59*, 1 (2007), 239–260.

[7] Balakrishnan, V. *Schaum's Outline of Theory and Problems of Graph Theory*. Schaum's Outline Series. McGraw-Hill, New York, USA, 1997.

[8] Battista, G. D., Eades, P., Tamassia, R., and Tollis, I. G. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, Upper Saddle River, New Jersey, USA, 1999.

[9] Bent, S. W., and Manber, U. On Non-intersecting Eulerian Circuits. *Discrete Applied Mathematics 18*, 1 (1987), 87–94.

[10] Bhatia, D., Mehtab, S., Krishnan, R., Indi, S. S., Basu, A., and Krishnan, Y. Icosahedral DNA Nanocapsules by Modular Assembly. *Angewandte Chemie International Edition 48*, 23 (2009), 4134–4137.

[11] Binns, C. *Introduction to Nanoscience and Nanotechnology*, vol. 14 of *Wiley Survival Guides in Engineering and Science*. John Wiley & Sons, Hoboken, N.J, USA, 2010.

[12] Boncheva, M., and Whitesides, G. M. Making Things by Self-assembly. *MRS Bulletin-Materials Research Society 30*, 10 (2005), 736.

[13] Bourke, P. PLY - Polygon File Format. `http://paulbourke.net/dataformats/ply/`, 2013. [Online; accessed 30-December-2013].

[14] Boyer, J. M., and Myrvold, W. J. On the Cutting Edge: Simplified $\mathcal{O}(n)$ Planarity by Edge Addition. *J. Graph Algorithms Appl. 8*, 2 (2004), 241–273.

[15] Castro, C. E., Kilchherr, F., Kim, D.-N., Shiao, E. L., Wauer, T., Wortmann, P., Bathe, M., and Dietz, H. A Primer to Scaffolded DNA Origami. *Nature methods 8*, 3 (2011), 221–229.

[16] Chen, J., and Seeman, N. C. Synthesis from DNA of a Molecule with the Connectivity of a Cube. *Nature 350*, 6319 (1991), 631–633.

[17] Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction To Algorithms*, second ed. MIT Press, Cambridge, Mass, 2001.

[18] Cromwell, P. R. *Polyhedra*. Cambridge University Press, Shaftesbury Rd, Cambridge, United Kingdom, 1999.

[19] Dairbekov, N., Alexandrov, A., Kutateladze, S., and Sossinsky, A. *Convex Polyhedra*. Springer, Berlin, Germany, 2005.

[20] Diestel, R. *Graph Theory*, second ed., vol. 173 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, USA, 2000.

[21] Douglas, S. M., Dietz, H., Liedl, T., Högberg, B., Graf, F., and Shih, W. M. Self-assembly of DNA into Nanoscale Three-dimensional Shapes. *Nature 459*, 7245 (2009), 414–418.

[22] Douglas, S. M., Marblestone, A. H., Teerapittayanon, S., Vazquez, A., Church, G. M., and Shih, W. M. Rapid Prototyping of 3D DNA-origami Shapes with caDNAno. *Nucleic Acids Research 37*, 15 (2009), 5001–5006.

[23] Døvling Andersen, L., and Fleischner, H. The NP-completeness of Finding A-trails in Eulerian graphs and of Finding Spanning Trees in Hypergraphs. *Discrete Applied Mathematics 59*, 3 (1995), 203–214.

[24] Edmonds, J., and Johnson, E. L. Matching, Euler Tours and the Chinese Postman. *Mathematical Programming 5*, 1 (1973), 88–124.

[25] Feldkamp, U., and Niemeyer, C. M. Rational Design of DNA Nanoarchitectures. *Angewandte Chemie International Edition 45*, 12 (2006), 1856–1876.

[26] Feynman, R. P. There is Plenty of Room at the Bottom. *Engineering and Science 23*, 5 (1960), 22–36.

[27] Fleischner, H. *Eulerian Graphs and Related Topics*, vol. 1 of *Annals of Discrete Mathematics 45*. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[28] Fu, T. J., and Seeman, N. C. DNA Double-crossover Molecules. *Biochemistry 32*, 13 (1993), 3211–3220.

[29] Goodman, R. P., Berry, R. M., and Turberfield, A. J. The Single-step Synthesis of a DNA Tetrahedron. *Chemical Communications*, 12 (2004), 1372–1373.

[30] Gould, R. J. Recent Advances on the Hamiltonian Problem: Survey III. *Graphs and Combinatorics 30*, 1 (2014), 1–46.

[31] Graver, J. E., Servatius, B., and Servatius, H. *Combinatorial Rigidity*, vol. 2 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, USA, 1993.

[32] Harary, F. *Graph Theory*. Addison-Wesley Series in Mathematics. Addison-Wesley Publishing Company Inc, Reading, MA, USA, 1969.

[33] He, Y., Ye, T., Su, M., Zhang, C., Ribbe, A. E., Jiang, W., and Mao, C. Hierarchical Self-assembly of DNA into Symmetric Supramolecular Polyhedra. *Nature 452*, 7184 (2008), 198–201.

[34] Högberg, B., and Gardell, J. vHelix - a Plugin for Autodesk Maya for DNA Nanostructure Design. `http://www.vhelix.net/`, 2013. [Online; accessed 30-December-2013].

[35] Hopcroft, J., and Tarjan, R. Efficient Planarity Testing. *Journal of the ACM (JACM) 21*, 4 (1974), 549–568.

[36] Johan Gardell, Pavan Kumar Areddy, A. L. B. K. K. G., and Högberg, B. vHelix for Maya – Lattice Free DNA Nanostructure CAD. Poster, FNANO, 2012.

[37] Jungnickel, D. *Graphs, Networks and Algorithms*, vol. 5 of *Algorithms and Computation in Mathematics*. Springer, Berlin, Germany, 1999.

[38] Kappraff, J. *Connections: The Geometric Bridge between Science and Art*, first ed. McGraw-Hill, Inc, New York, USA, 1991.

[39] Kolmogorov, V. Blossom V: A New Implementation of a Minimum Cost Perfect Matching Algorithm. *Mathematical Programming Computation 1*, 1 (2009), 43–67.

[40] Lo, P. K., Metera, K. L., and Sleiman, H. F. Self-assembly of Three-dimensional DNA Nanostructures and Potential Biological Applications. *Current Opinion in Chemical Biology 14*, 5 (2010), 597–607.

[41] Meyer, W. A. *Geometry and its Applications*. Elsevier Academic Press, Burlington, MA, USA, 2006.

[42] Qiu, H., Dewan, J. C., and Seeman, N. C. A DNA Decamer with a Sticky-end: The Crystal Structure of d-CGACGATCGT. *Journal of Molecular Biology 267*, 4 (1997), 881–898.

[43] Richmond, T. J., and Davey, C. A. The Structure of DNA in the Nucleosome Core. *Nature 423*, 6936 (2003), 145–150.

[44] Robinson, B. H., and Seeman, N. C. The Design of a Biochip: a Self-assembling Molecular-scale Memory Device. *Protein Engineering 1*, 4 (1987), 295–300.

[45] Rothemund, P. W. Design of DNA Origami. In *Proceedings of the 2005 IEEE/ACM International Conference on Computer-aided Design* (2005), IEEE Computer Society, pp. 471–478.

[46] Rothemund, P. W. Folding DNA to Create Nanoscale Shapes and Patterns. *Nature 440*, 7082 (2006), 297–302.

[47] Rothemund, P. W. Scaffolded DNA Origami: From Generalized Multicrossovers to Polygonal Networks. In *Nanotechnology: Science and Computation*. Springer, 2006, pp. 3–21.

[48] Sa-Ardyen, P., Vologodskii, A. V., and Seeman, N. C. The Flexibility of DNA Double Crossover Molecules. *Biophysical Journal 84*, 6 (2003), 3829–3837.

[49] Seeman, N. C. Nucleic Acid Junctions and Lattices. *Journal of Theoretical Biology 99*, 2 (1982), 237–247.

[50] Seeman, N. C. An Overview of Structural DNA Nanotechnology. *Molecular Biotechnology 37*, 3 (2007), 246–257.

[51] Seeman, N. C. Nanomaterials based on DNA. *Annual Review of biochemistry 79* (2010), 65.

[52] Shih, W. M., and Lin, C. Knitting Complex Weaves with DNA Origami. *Current Opinion in Structural Biology 20*, 3 (2010), 276–282.

[53] Shih, W. M., Quispe, J. D., and Joyce, G. F. A 1.7-kilobase Single-stranded DNA that Folds into a Nanoscale Octahedron. *Nature 427*, 6975 (2004), 618–621.

[54] Shlyakhtenko, L. S., Potaman, V. N., Sinden, R. R., Gall, A. A., and Lyubchenko, Y. L. Structure and Dynamics of Three-way DNA Junctions: Atomic Force Microscopy Studies. *Nucleic Acids Research 28*, 18 (2000), 3472–3477.

[55] Siek, J. MutableGraph - 1.54.0. `http://www.boost.org/doc/libs/1_54_0/libs/graph/doc/MutableGraph.html`, 2001. [Online; accessed 29-December-2013].

[56] Thai, M., and Sahni, S. *Computing and Combinatorics: 16th Annual International Conference, COCOON 2010, Nha Trang, Vietnam, July 19-21, 2010 Proceedings*. Springer, 2010.

[57] Tian, Y., He, Y., Chen, Y., Yin, P., and Mao, C. A DNAzyme That Walks Processively and Autonomously along a One-Dimensional Track. *Angewandte Chemie International Edition 44*, 28.

[58] Tsai, M.-T., and West, D. B. A New Proof of 3-colorability of Eulerian Triangulations. *Ars Mathematica Contemporanea 4*, 1 (2011).

[59] West, D. B. *Introduction to Graph Theory*, 2nd ed. Prentice Hall, Upper Saddle River, New Jersey, USA, 2001.

[60] Wilson, R. J. *Introduction to Graph Theory*, 4th ed. Prentice Hall, Harlow, Essex, England, 1996.

[61] Windsor, A. Boost Graph Library: Boyer-Myrvold Planarity Testing/Embedding - 1.36.0. `http://www.boost.org/doc/libs/1_36_0/libs/graph/doc/boyer_myrvold.html`, 2007. [Online; accessed 29-December-2013].

[62] Winfree, E. Simulations of Computing by Self-assembly. Tech. Rep. 22, California Institute of Technology, Pasadena, California, 1998.

[63] Winfree, E., Liu, F., Wenzler, L. A., and Seeman, N. C. Design and Self-assembly of Two-dimensional DNA Crystals. *Nature 394*, 6693 (1998), 539–544.

[64] Wolf, E. L. *Nanophysics and Nanotechnology: An Introduction to Modern Concepts in Nanoscience*, second ed. WILEY-VCH Verlag GmbH & Co, Weinheim, Germany, 2008.

[65] Zhang, Y., and Seeman, N. C. Construction of a DNA-truncated Octahedron. *Journal of the American Chemical Society 116*, 5 (1994), 1661–1669.

[66] Ziegler, G. *Lectures on Polytopes*, vol. 152 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, USA, 1995.

Chapter 7

Appendix

The BScOR software package is a pipelined set of executables for finding a scaffold routing path on a (triangular) 3D mesh given as a Polgon File Format (PLY). The BScOR pipeline is illustrated in Figure 7.1. The first executable is a PLY to DIMACS converter, which extracts the graph adjacency information from the objects surface mesh. A surface mesh gives an approximation of a 3D object by a set of polygons. A PLY text file [13] gives a description of each polygon in the mesh. The converter outputs a graph in DIMACS format.

Given that the graph may not be Eulerian, the second executable constructs a weighted complete graph on the odd degree vertices which is then output as a DIMACS file. The DIMACS file, containing edge weights corresponding to the length of shortest paths between odd degree vertices, is then fed to Kolmogorov's Blossom V implementation [39] for the minimum weight perfect matching problem. The output of Blossom V in the BScOR pipeline is a file with the minimum weight perfect matching.

The original graph and the matching file are then given as an input to the fourth executable which adds multiedges based on the matching. The resulting Eulerian graph is then given as an input to an executable utilizing the Boyer Myrvold planar embedding algorithm. The executable outputs a custom text file which stores the cyclic order of edges around the vertices according to the planar embedding. Finally, the last executable searches for an A-trail for the given embedding. Additional Matlab scripts have also been developed for visualizing the scaffold routing path given the resulting trail file and the PLY file for the 3D mesh.

BScOR was implemented in standard `C++` and uses the boost graph library. BScOR has been compiled by GNU `C++` compiler (`g++` version 4.6.3), and has been tested on 64 bit Linux based machines.

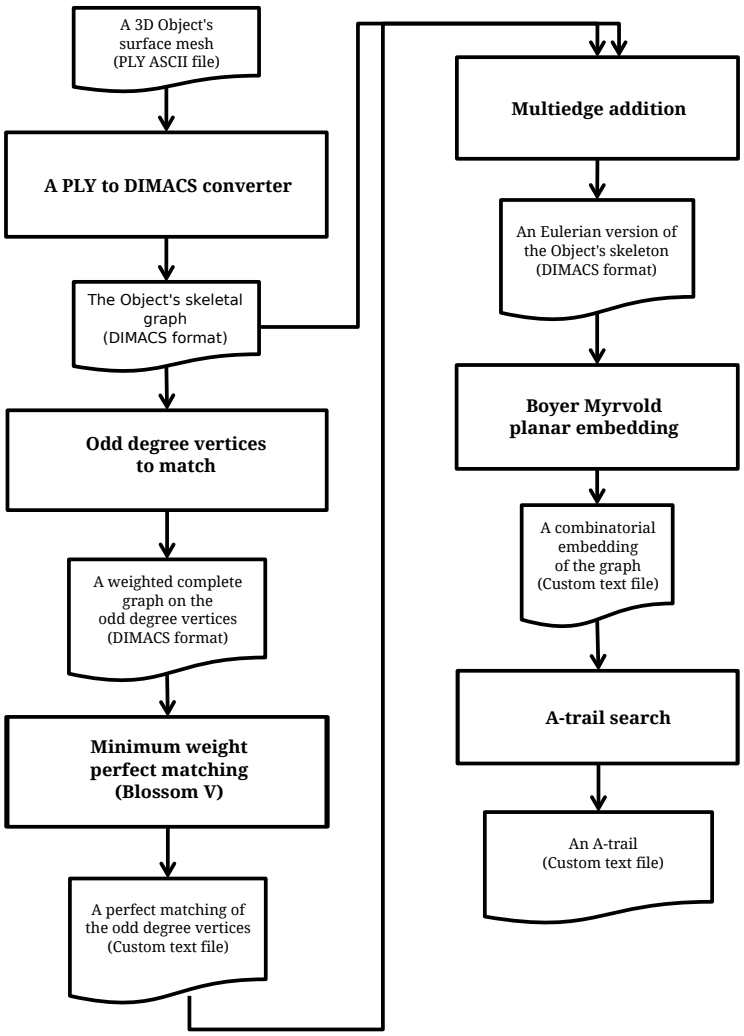


Figure 7.1: The BScOR pipeline.