

Algorithmic Folding  
Hasso-Plattner-Institute  
In-Depth Homework Assignment Project Report  
End-to-End Vertex Unfolding

Franziska Lauterbach  
Student ID: 811668

February 2023

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Algorithm</b>	<b>2</b>
<b>3</b>	<b>Implementation</b>	<b>3</b>
<b>4</b>	<b>Challenges</b>	<b>5</b>

# 1 Introduction

This report is part of the practical part of the “Algorithmic Folding” at the HPI in the winterterm 2022/2023. The corresponding code can be found on GitHub (<https://github.com/NessaCat/vertex-unfolding-assignment>). The topic of the assignment is “Vertex Unfolding”.

## 2 Algorithm

When it comes to unfolding polyhedra, there are different kinds of approaches with different limitations and rules. *Edge Unfolding* limits cutting to the edges of a polyhedron, while *General Unfolding* does not restrict cuttings and just aims for a single, non-overlapping unfolding. *Vertex Unfolding* on the other hand, unfolds a polyhedron along its vertices and lays it out in a chain. Relevant for this assignment is the algorithm proposed by *Demaine, Eppstein, Erickson, Hart and O'Rourke* that was presented in the lecture.

The algorithm consists of 10 steps.

1. triangulation of mesh
2. creating a dual graph
3. finding a spanning tree

As the algorithm only on triangulated mesh only, the first step ensured that constraint. After the triangulation has happened, a dual graph is created. The dual graph has a vertex for every triangle and connects two vertices whenever the triangles share an edge. Starting from the dual graph, a spanning tree is identified.

4. identify “level-1 ears”
5. identify “level-2 ears”
6. repeat
7. replace with cycle

After *pseudo-unfolding* the spanning tree, so called ears, of different levels, can be identified. An example of an ear can be seen in figure 1. Based on the ears, a so-called *facet-path* can then be constructed. A facet path is path alternating between vertices and faces. When building the facet path, the goal is to achieve a cycle. There are 5 types of ear constellations that must be considered and not for all of them, a cycle can be achieved. For this assignment, the focus was on implementing the path for all types and not just the one case provided in the lecture.

8. switch connections

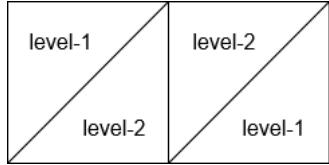


Figure 1: Example of Ears

9. take *Euler tour*

10. lay out unfolding

After finishing the facet paths, it is not guaranteed for them to be connected. Therefore, step 8 is needed to switch some of the connections to provide a *Eulerian Circuit*. The *Euler tour* along the facet path, that determines the order of the faces when laying them out in step 10.

### 3 Implementation

As mentioned before, the first focus was to implement the facet path construction of all types of ear configurations presented in the lecture. An overview of those can be seen in figure 2.

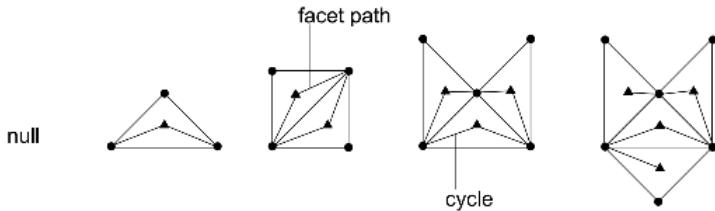


Figure 2: Ear configurations and respective facet path

The implementation was done using *Python* inside a *Jupyter Notebook*. The code is build heavily upon the provided code in the lecture.

First, a set of faces and their spanning tree was manually created for each type, excluding the *null case*. As provided by the course, an array is filled with the *ear-level* of each face (or: node, as we are operating on a spanning tree).

With the information of the levels, we can start differentiating the different types and building their respective facet path. For the first case of just one single face, this is a rather simple task. The facet path consists of two edges, one from a vertex to the face and another from the face to another vertex. In the code, this example only works on a minimal example of one given face.

```
if len(level_array) == 1:
```

```

facet_graph.add_edge(0, faces[0][0])
facet_graph.add_edge(0, faces[0][1])

```

Identifying the other types can be done by counting the number of neighboring faces of a level-2 ear. As the faces are triangles, this number can be at most 3. For each case, a different facet path is needed. For 1 neighbor, there are two vertices shared by the two triangles. Those will be used to create the cyclic facet path.

```

if len(relevant_neighbours) == 1:
    common_vertices = np.intersect1d(faces[face_id],
                                    faces[relevant_neighbours[0]])
    facet_graph.add_edge(face_id, common_vertices[0])
    facet_graph.add_edge(face_id, common_vertices[1])

    facet_graph.add_edge(relevant_neighbours[0],
                        common_vertices[0])
    facet_graph.add_edge(relevant_neighbours[0],
                        common_vertices[1])

```

In case of two neighbours, all three triangles share one vertex and each level-1 neighbor shares one additional vertex with the level-2 ear. This one vertex must be identified to successfully construct a cyclic facet path.

```

if len(relevant_neighbours) == 2:
    common_vertices_0 = np.intersect1d(faces[face_id],
                                    faces[relevant_neighbours[0]])
    common_vertices_1 = np.intersect1d(faces[face_id],
                                    faces[relevant_neighbours[1]])
    common_vertex = np.intersect1d(common_vertices_0,
                                common_vertices_1)
    common_vertices_0 = np.setdiff1d(common_vertices_0,
                                common_vertex)
    common_vertices_1 = np.setdiff1d(common_vertices_1,
                                common_vertex)

    facet_graph.add_edge(face_id, common_vertices_0[0])
    facet_graph.add_edge(face_id, common_vertices_1[0])

    facet_graph.add_edge(common_vertices_0[0],
                        relevant_neighbours[0])
    facet_graph.add_edge(common_vertices_1[0],
                        relevant_neighbours[1])

    facet_graph.add_edge(relevant_neighbours[0],
                        common_vertex[0])
    facet_graph.add_edge(relevant_neighbours[1],
                        common_vertex[0])

```

For the last type, a cyclic facet path is not possible and those constructions must be handled carefully later when switching connections to take an Euler tour.

```

if len(relevant_neighbours) == 3:
    start_face = relevant_neighbours[0]
    common_start = np.intersect1d(faces[start_face],
                                  faces[face_id])
    facet_graph.add_edge(start_face,
                          common_start[0])

    facet_graph.add_edge(common_start[0],
                          face_id)
    facet_graph.add_edge(face_id,
                          common_start[1])

if common_start[1] in faces[relevant_neighbours[1]]:
    second_face = relevant_neighbours[1]

    last_face = relevant_neighbours[2]
else:
    second_face = relevant_neighbours[2]
    last_face = relevant_neighbours[1]

last_vertex = np.intersect1d(faces[second_face],
                            faces[last_face])[0]

facet_graph.add_edge(common_start[1],
                      second_face)
facet_graph.add_edge(second_face,
                      last_vertex)
facet_graph.add_edge(last_vertex,
                      last_face)

```

## 4 Challenges

The biggest challenge was missing knowledge and experience when it comes to programmatically dealing with unfoldings. I tried to overcome it by only focusing on the base types presented in the lecture and only using a minimal set of faces required for the respective type. Another great challenge was coming up with test data for the implementation, so all testing was unfortunately done on a single, minimal example. Another big challenge would be the actual laying out of the triangles in a non-overlapping manner, but I did not try to master this challenge (yet).