University of Zurich[UZH]

Lars Zawallich

Unfolding Polyhedra via Tabu Search

TECHNICAL REPORT – No. IFI-2023.03

October 2023

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

L. Zawallich: Unfolding Polyhedra via Tabu Search
Technical Report No. IFI-2023.03, October 2023
Visualization and Multimedia Lab
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland
URL: https://ifi.uzh.ch/vmml

Technical Report IFI-2023.03, Department of Informatics, University of Zurich, 2023

Unfolding Polyhedra via Tabu Search

Lars Zawallich

Abstract Folding a discrete geometry from a flat sheet of material is one way to construct a 3D object. While nowadays for this purpose a lot of attention lays on 3D printing, folding can be a considerable alternative, complementing the possibilities 3D printing provides. A typical creation pipeline first designs the 3D object, unfolds it, prints and cuts the unfold pattern from a 2D material, and then refolds the object. Within this work we focus on the unfold part of this pipeline. Most current unfolding approaches segment the input, which has structural downsides for the refolded result. Therefore, we are aiming to unfold the input into a single-patched pattern. Our algorithm applies tabu search to the topic of unfolding. We show empirically that our algorithm is faster and more reliable than other methods unfolding into single-patched unfold-patterns. Moreover, our algorithm can handle any sort of flat polygon as faces, while comparable methods are bound to triangles.

Keywords Unfolding · Papercraft · Discrete Optimization · Computational Geometry

1 Introduction

Folding paper has a long history. The oldest known folded piece of papyrus dates back to ancient Egypt and is a folded road map [19]. Even though it is not certain that folding paper as an art was invented in Japan, the oldest known origami was created there in the 6th century [15]. The first records dealing with the unfolding of polyhedra date back to Albrecht Dürer in 1525 [8]. Today, folding and unfolding can be found in many aspects of our lives. Be it a paper plane, an origami, arts and crafts class in school, packaging, or architectural prototyping, folding or unfolding is involved in one way or another. A recent and prominent example of folding and unfolding is the James Webb Space Telescope, which had to be packed to fit into the delivering rocket and then unfolded in space.

Within the area of digital fabrication, papercraft only represents a small part of the overall field. Instead, 3D printers have received a lot of attention, especially in the past years [17]. Despite all the advantages of 3D printing, there are some disadvantages to consider. For instance, most materials used are not renewable, the printers are more expensive and printing times are longer in comparison to 2D printers.

In contrast, paper is more eco-friendly, 2D printers are widely available and even printers for very large paper sizes can be found in many copy shops. Additionally, the resulting object is very light. If more stability is needed, cardboard or other more stiff materials can be used. On top of that, self-folding allows for automation with minimal manual work, which can result in cheaper, quicker and easier-to-transport production methods compared to classical 3D creation pipelines [9]. Instead of replacing 3D printing, we argue that models folded from paper can augment the possibilities 3D printing offers.

Unfortunately, the problem of unfolding polyhedra is complex and to this day, it is not even known if every polyhedron can be unfolded at all [6]. On top of that, it is known that non-convex polyhedra, the most common type in real world applications, can not always be unfolded via edge unfolding, the most intuitive unfolding technique. Current approaches try to overcome this issue by segmenting the unfolding into separate parts, which are unfoldable. In the most extreme case, this can lead to "unfoldings" consisting of numerous segments each containing a single or very few faces. While

L. Zawallich (LarsZawallich@gmail.com)
University of Zurich, Zurich, Switzerland

it is proven that non-convex polyhedra are not edge-unfoldable, in practice that issue rarely occurs.

Within this work, we present a simple algorithm which edge-unfolds a given polyhedron fast and reliably – given it is unfoldable. Our technique is more reliable and orders of magnitudes faster than any comparable method. Moreover, in contrast to comparable methods, our approach does not need any special starting point for its optimization.

2 Background

Our algorithm is a tabu search algorithm [10], which has been applied to edge unfolding. In the following, we briefly review the basics for unfolding as well as the tabu search algorithm.

2.1 Unfolding

The following definitions are mostly covered in the book *Geometric Folding Algorithms: Linkages, Origami, Polyhedra* [6].
There are two main ways to unfold a polyhedron:

1. Edge unfolding
2. General unfolding

Edge unfolding only allows cutting the polyhedron along its given edges. General unfolding allows arbitrary cuts. After cutting, both approaches are then unfolding the cut-open polyhedron along its edges, to flatten it into a plane. When using edge unfolding the number of cuts and folds is bound from above by the number of edges in the polyhedron. When using general unfolding, the number of cuts and folds can become arbitrarily high, resulting in tedious work when refolding. One famous edge unfolding technique is *Steepest Edge Unfolding* [21]. A general unfolding technique is *Star-Unfolding* [6, Chapter 24.3]. In this work, we will only use edge unfolding as a technique.

To this day, it is unknown if all polyhedra can be unfolded. An overview of this issue can be seen in Table 1.

	Edge unfolding	**General unfolding**
Convex	**?**	✓
Non-convex	✗	**?**

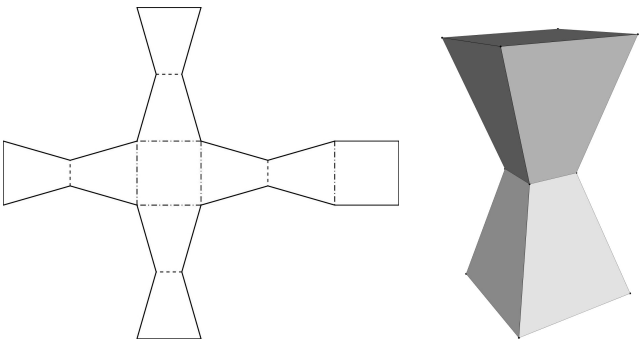
Table 1: Status of main questions concerning non-overlapping unfoldings; **?** – Unknown; ✓ – Always possible; ✗ – Known counter-examples; [6, Table 22.1]

To unfold every face of a polyhedron, generally, at least one cut is required at each vertex. The only exception occurs at vertices with incident angles of 2π, i.e. at vertices with a Gaussian curvature of zero. In this case, no cut is needed to unfold the surrounding faces into a plane. Doing the cut nevertheless has the advantage of being able to separate the coplanar faces in the unfolding. This separation can be advantageous in resolving overlaps later on. For genus zero polyhedra, the cut edges must form a spanning tree over its vertices. If the cut edges would not form a spanning tree, either some vertices were not reached, or the tree had a cycle. In the first case, the faces around this vertex would not be unfoldable into a plane – except for the above described case. In the latter case the faces enclosed by the cycle would be cut out completely, disconnecting it from the unfolding. Such a spanning tree formed by the cut edges is called a *cut-tree*.

A single-patched overlap-free unfolding created by using edge unfolding is called a *net*. Rarely, this term is also used to name an overlap-free unfolding generated by general unfolding. To avoid confusion, we will only use it in relation to edge unfolding.

The net of an unfolded polyhedron itself is a polygon. This polygon needs some indications on where to fold it and in which direction, to regain the original polyhedron. We follow the convention of dashed lines representing valley folds and dash-dotted lines representing mountain folds. Figure 1 shows an example. To prevent overloading of our visualizations, we only use these fold-representations when it comes to folding.

If a polyhedron is not edge-unfoldable without overlap, we will call it *not-unfoldable*. In the literature, this term is also referred to as *ununfoldable*.



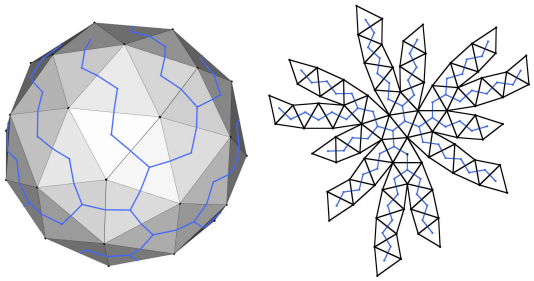
(a) The folding pattern of a sand clock shape. Dashed lines represent valley folds and dash-dotted lines represent mountain folds.

(b) The corresponding 3D shape.

Fig. 1: An example for line-styles representing different folding directions.

There are different ways to look at an unfolding. One is to define it via its cutting of the graph (e.g. [24]). Another way is to define the unfolding as a spanning-tree of the dual-graph of the mesh (e.g. [11]). While any approach working on the dual-graph will always have to find a spanning-tree, approaches working on graphs can only work with cut-trees in the genus zero case. With higher genuses the cutting needs to be a *cut-graph* – instead of a cut-tree – with as many cycles as the polyhedron's genus.

In this work, we use the approach of spanning-trees of dual-graphs. Figure 2 shows an example of such a spanning-tree. In the context of unfolding, we call a spanning-tree of the dual-graph an *unfold-tree*.



(a) A folded icosphere. (b) The corresponding unfolding.

Fig. 2: A folded and unfolded icosphere with 80 faces. The unfold-tree is visualized in blue.

2.2 Tabu Search

Tabu Search is a metaheuristic for optimization and has first been introduced by Fred Glover [10]. The basic technique is outlined in Algorithm 1.

Algorithm 1 Tabu Search

```
function TABUSEARCH(x⃗, maxTabuSize)
                          ▷ x⃗ is an initial state of parameters
    tabuList = list()
    while !stoppingCriterion() do
        neighbors ← getNeighbors(x⃗)
        neighbors ← removeTabus(neighbors, tabuList)
                                  ▷ Avoids undoing past steps
        bestNeighbor ← neighbors.best()
        tabuList.push(x⃗)
        x⃗ ← bestNeighbor
        tabuList.shrinkToSize(maxTabuSize)
                                ▷ Removes oldest entries first
    end while
end function
```

As with other combinatorial optimization problems, computing derivatives is hard to impossible in the context of unfolding. Therefore, many known and prominent optimization techniques (like gradient descent, or Newton's method) can not be applied. Tabu search does not require any sort of derivative. While other derivative free optimization techniques in their basic variant, like hill climbing [16], are unable to overcome local minima, tabu search is able to do so. This is done by accepting the locally best neighbor instead of only accepting improving neighbors. By then memorizing the past states, the algorithm prevents falling back into a local minimum. Two possible issues tabu search poses are to determine when the optimum is reached and how many past steps to remember. These two issues are addressed in Sections 4 and 4.4.

3 Related Work

Besides the purpose of creating paper models or art, folding and unfolding polyhedra appears in other areas of research as well. In robotics, a recent publication suggested reconfiguring modular robots with folding and unfolding techniques [29]. Also in the field of robotics, the review by Rus et al. gives a great overview of so-called origami robots, which are created from a flat material via folding [20].

Solving the big questions in unfolding (see Table 1) turned out to be very hard. Instead, current works in this area focus on special cases like orthogonal polyhedra [3, 4], or edge unzipping [5].

One practical approach to create papercraft models was presented by Tachi, who proposed to lay out the faces of a mesh into a plane, connecting them with so-called tucking-molecules [25]. These molecules get folded into the body of the result, making them invisible from the outside. This approach does not need to cut any paper, but only fold, which can be advantageous. The approach has been extended in the work by Demain et al. to use less filling-material, rendering the algorithm more practical [7].

Another practical approach of creating papercraft models are developable surfaces. The core idea is to find developable patches – which means they have a zero Gaussian curvature in each point – representing the input mesh as well as possible. Then, these patches are cut out of paper (or another developable material) and attached together. This attachment may involve a bending of the patches. Different approaches include e.g. optimizing cut-lines for strips, which then are made developable [18], approximations with cones and cylinders [22], optimizing for hinges [23], or minimizing the number and complexity of patches while keeping the

approximation error low [13]. Developable surfaces can be seen as a segmentation technique, which works with bends instead of folds.

When working with folds, many practical approaches (see next paragraphs) favor edge unfolding over general unfolding. Unfortunately, for edge unfolding there are known polyhedra, which cannot be edge-unfolded. To overcome this issue, many approaches choose to segment the unfolding into parts.

Straub et al. explored different heuristics to find cut-trees by assigning a value to the edges of the mesh and then finding a minimal spanning tree [24]. Their approach to remove overlaps is to cut the unfolding into several parts [24, Section 2.2], which is a segmentation. Instead of finding cut-trees Haenselmann et al. explored different heuristics for spanning a tree over the dual-graph of the mesh, which is equivalent to laying out its faces in a specific order [11]. Takahashi et al. proposed to start off with small patches and stitch them together, using a genetic optimization algorithm [26]. While they are trying to minimize the number of segments, it is still possible and common for their algorithm to yield a segmented result.

Instead of accepting segmentation as a necessity, Xi et al. segmented a given mesh by analyzing overlaps in unfoldings created by an easy-to-compute method [27]. The resulting segments can then be unfolded without self-overlap via the easy-to-compute method they used in their pipeline. Additionally, they considered the continuous foldability of the unfolding, which is important for e.g. self-folding.

The topics of self-foldability and continuous unfolding have gained more interest in the past years, since it is an important concept to automatically create robots and structures from 2D shapes, by exposing them to e.g. heat [2]. This possibility motivated recent publications (e.g. [12, 28]) to focus on the continuous and self-folding property of unfoldings, rather than the question of unfoldability itself.

To the best knowledge of the authors, there are only two published approaches aiming to edge unfold a given non-convex mesh into a single-patched unfolding. One is the aforementioned approach by Takahashi et al. which aims to create a single patched unfolding, but accepts segmented results, if no single patch can be found [26]. The other approach uses simulated annealing to unfold a given mesh, while additionally considering gluetags [14]. In our experiemnts, this approach scales poorly and lacks reliability (see Figures 6 and 7).

Our method improves on all aforementioned shortcomings current approaches have. For example, it scales better, while also not relying on segmentation.

4 Methods

Unfolding a polyhedron can be seen as an optimization problem. As such, the function f to minimize takes the current unfold configuration as an input and yields the number of overlapping faces. Obviously, f will always yield a non-negative integer and its minimum is reached with $f(\vec{x}) = 0$. Knowing the global minimum is especially advantageous for optimization. Instead of hoping to have reached a global minimum, it is very easy to determine if a given minimum is local or global.

The algorithm presented in this article applies tabu search to the topic of unfolding. In particular, the algorithm can be split into the following parts:

1. The input (Sections 4.1)
2. An initial unfolder (Section 4.2)
3. A strategy to select the best step (Section 4.3)
4. A strategy to overcome local minima (Sections 4.4 and 4.5)
5. (Optional) Additional optimization parameters (Section 4.6)

Additionally, in Section 4.7 efficient overlap detection, and in Section 4.8 data structures are discussed.

4.1 Input

As input, our method takes an orientable mesh. This mesh does not need to be triangular, but for the unfolding every face needs to be planar. To allow for an input without planar faces, we implemented the planarization flow presented by Alexa and Wardetzky [1]. The planarization is done as a preprocessing step and is independent of the unfolding algorithm presented in this paper. Besides the planarity of faces, the mesh has to be a manifold. There are no constraints on the genus, convexity or other remaining properties of the mesh. In theory, the mesh could even self-intersect. This would make re-folding the result hard to impossible, but does not hinder our algorithm from producing a valid result.

4.2 The Initial Unfolder

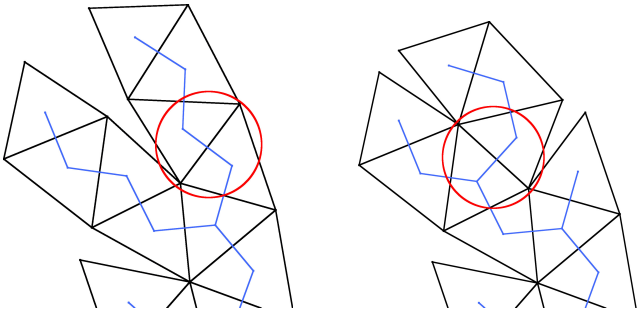
For the initial unfolding it is acceptable to contain overlaps. Furthermore, different initial unfoldings are needed, to explore different areas of the unfolding space. Therefore, we aimed for a non-deterministic algorithm quickly producing different unfoldings, which may contain overlaps.

Our choice was the *Steepest Edge Unfolding* algorithm [21]. Originally designed to unfold convex polyhedra, it is also applicable to non-convex polyhedra. The

resulting unfoldings will very likely contain overlaps, but that is acceptable, as mentioned above. Since the Steepest Edge Unfolding uses a randomized direction as a cut direction, it is able to create different unfoldings for the same polyhedron. Moreover, the time complexity is bound by $\mathcal{O}(V + F)$ (one cut per vertex, plus setting up the unfold-tree, which scales linearly with the number of faces). Finally, the resulting unfoldings tend to have fewer overlaps than other methods with comparable time complexity, like a random unfolding, yield.

4.3 Selecting The Best Step

We restrict the neighborhood search of our algorithm to one parameter at a time. In particular, the algorithm picks a random overlapping face and tries to attach it to another possible neighbor in the dual-graph. For readability, the procedure of attaching a face to another neighbor in the dual-graph will be called a *move*. An example for a move can be seen in Figure 3. The goal of our algorithm is to move faces, such that the total number of overlapping faces decreases. Please note, that this does not necessarily mean a face needs to be overlap-free after a move. Since each non-leaf face in the unfold-tree represents a subtree, moving that face corresponds to moving the subtree as a whole. Therefore, the moved face may still overlap, while the total number of overlaps of the subtree has been reduced.



(a) Before attaching a node to a new parent.

(b) After attaching a node to a new parent.

Fig. 3: Attaching a node in the unfolding to a new parent-node in the unfold-tree. The spanning-tree is visualized in blue.

It may happen that the randomly selected face can not be attached to a new parent-node. This can occur for several reasons. The two main reasons are, there is no new parent-node available, or the move would create more overlaps than the current state has. The first case occurs for example when a face is fully connected, all possible dual-graph neighbors are located in its own subtree (see Figure 4), or all possible dual-graph neighbors are in the tabu list (see Section 4.4). If a face can not be attached to a new parent-node, our algorithm recursively climbs up the tree and tries to move the parent-node of the initially selected one instead, until the root node is reached.

4.4 Local Minima

Tabu search stores m past steps in a so-called tabu list. Any step in the tabu list can not be undone. Obviously, the value of m is critical for this algorithm. A value that is too large would make the algorithm block every possible move, resulting in no direction left to go. For example, if only two faces are overlapping, which are both close to the root node (e.g. both having a distance of two to the root node), there are at maximum eight possible moves. If m was greater than eight, it is possible that all moves are blocked by the tabu list. While too large values for m may block every possible move, a value that is too small would result in the algorithm to fall back into local minima.

In our work, we used

$$m = val \cdot \log_{val}(|F|)$$

throughout. val is the average valence in the dual-graph of the mesh and $|F|$ is the number of faces. For triangle meshes the valence of each node representing a face, which is not located at a border, is 3 and thus $val = 3$ on average as well. The minimal height of a tree grows logarithmically with the number of its entries. Degenerate cases exist, but it is very unlikely to create one randomly. But even if such a case occurs, it is even more unlikely for the whole degenerate tree to be one local minimum. Still, in such a case, higher values for m (up to the size of the configuration space) might be necessary. The val term multiplies the height of a tree with the number of neighbors it can be applied to. Thus, $val \cdot \log_{val}(|F|)$ is the number of possible moves one branch in a full tree can perform on average. This value statistically allows the algorithm to test every possible move on a branch, before it can undo moves from that branch.

Generally, it is better to slightly overshoot the value of m by a bit, since it is easier to determine if the solver can not perform any moves, compared to detecting if the solver falls back into a local minimum. Therefore, we are aiming for a reasonable upper bound of steps to block, rather than an exact value. It can always happen that the solver gets stuck in a situation where every possible move gets blocked by filtering out previous moves (see the example from above). Detecting such a

case is easy: If no move can be performed at all anymore, the tabu list gets cleared, enabling the solver to perform moves again. Empirically, we determined that this situation occurs very rarely.

4.5 Switching Root Nodes

In our data structure (Section 4.8), the root node of the unfold-tree is not movable to a new parent. This can lead to constellations, where the solver gets stuck. Such a constellation could occur if two nodes overlap and all dual-graph neighbors are located in their own sub-trees. In such a situation, the overlapping nodes can not get attached to any neighbor, since such a move would cut the tree into two. Also, if the nodes are high up in the unfold-tree, moving their parent nodes might not help either. Such a case could be resolved by moving non-overlapping nodes up and reordering the tree, which is a very time-consuming operation. An example for such a case is illustrated in Figure 4.

Instead of solving this issue directly, we implement a *reroot* method, which selects a new root node, when such a situation is detected. From an implementation point of view, rerooting is done by climbing up the tree from the new root node to the current one and inverting each parent link, the transformation, as well as the respective child entry in each node. When the solver gets stuck in a situation like described above, our algorithm reroots to a node located in the subtree of the overlapping node. That way, the climbing direction from this node is inverted and all dual-graph neighbors which were unreachable before are now possible to move to.

The goal of our algorithm is to find a net of a polyhedron and not a specific tree-structure. Therefore, we are free to switch root nodes. While the tree structure is needed to compute the unfolding, a fixed root also causes issues. E.g. when the root node is overlapping another node, the issue might be easy to solve by moving the root node to another position (see Figure 4). But because the root node is not movable, in such a situation the overlap needs to be solved by moving the rest of the tree around the root node. Such an operation is very difficult to perform for a highly randomized algorithm like ours. Moreover, it is practically impossible to detect such a situation and its correct solution. Since the whole idea of our algorithm is to be built from simple but robust parts, detecting these complex situations or even their solutions would violate this major construction philosophy. Instead, we make use of an important observation: One unfold-pattern can be created by many isomorphic unfold-trees.

Therefore, to prevent any issue related to a fixed root node, we randomly reroot in every iteration, except if we detect a blocking situation like described in the first paragraph of this section. With this strategy, we mimic the behavior of an unfold-pattern, which can perform any move, while still maintaining the simple tree-structure and all its benefits.

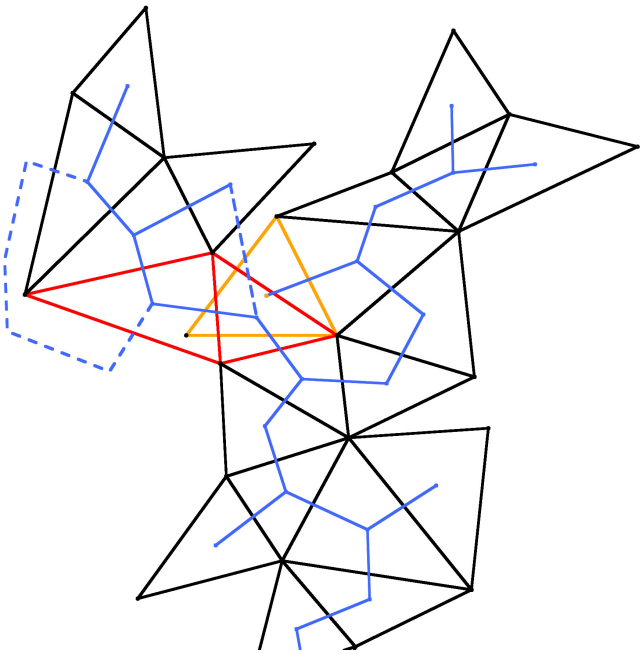


Fig. 4: A blocking situation where no colliding node or parent node can be moved anymore. The root node is highlighted in orange, the unfold-tree is marked with blue lines and overlapping faces are marked in red. In this case, the red faces are overlapping the root node. The root node is overlapping as well, but for visibility is not marked in red. Both overlapping faces have their only remaining dual-graph neighbor within their own subtree. These dual-graph neighbors are connected to the nodes via blue dashed lines. Moving either of the overlapping faces to their dual-graph neighbor would disconnect them from the root node.

4.6 Optional Optimization Parameters

In its basic version our algorithm “just” unfolds polyhedra and accepts the first output that is overlap-free. For some polyhedra more than one net exists, though. Our algorithm can be extended by additional optimization parameters, to enforce other constraints on the resulting unfolding than just being overlap-free. For example, a certain aspect ratio may be desired, to better fit standard paper sizes.

Such additional constraints can be implemented by adding them to the optimization function f. It is important for each constraint to have a well known and

reachable minimum. The stopping criterion then needs to be adjusted to the sum of minimums of all constraints. Moreover, the selection strategy discussed in Section 4.3 needs to be extended as well, to allow selecting faces violating additional constraints. Lastly, the memory discussed in Section 4.4 needs to be adjusted appropriately.

When applying constraints, each constraint is guaranteed to be met, but the runtime of the algorithm can worsen arbitrarily. Furthermore, there is no guarantee that any constraint is combinable with being overlap-free. Therefore, great care is advised when working with constraints.

4.7 Efficient Overlap Detection

After each change in the unfold-tree, it is necessary to determine the resulting number of overlaps. This is by far the most time-consuming part of the whole algorithm. Therefore, it is crucial to implement this part as efficiently as possible. In a naive implementation, detecting all overlaps would pose a time complexity of $\mathcal{O}(n^2)$. Please note that in the worst case, if every face overlaps every other face, the time complexity can not be better than $\mathcal{O}(n^2)$. To improve the average speed, we decided to implement a sweep line algorithm, which reduces the average time complexity down to $\mathcal{O}(n \log n)$. The algorithm is described in Algorithm 2.

Algorithm 2 Sweep Line Overlap Detection

```
Require: p⃗                          ▷ p⃗ is a list of unfolded polygons
  function DETECTOVERLAPS(p⃗)
    p⃗' ← sort(p⃗, d⃗)
    ▷ Sort by lower bounding box coordinate in dimension d⃗
    overlaps ← 0
    for i ← 0 to |p⃗'| do
      f_i ← p⃗'_i
      for j ← i + 1 to |p⃗'| do
        f_j ← p⃗'_j
        if f_j.bBox.lower_d⃗() > f_i.bBox.upper_d⃗() then
          break
        end if
        if f_i.intersects(f_j) then
          overlaps ← overlaps + 1
        end if
      end for
    end for
    return overlaps
  end function
```

Since our algorithm only changes a part of the unfolding at a time, it would be a waste to compute all overlaps again after moving a small subtree. Whenever the moved subtree contains less than $\log n$ polygons, we perform a naive overlap test on these polygons. If the number of moved polygons is larger, we recompute all overlaps using the sweepline algorithm.

4.8 Data Structures

In our implementation, we are using a list of vertices and a list of face indices as mesh data structure. Since our method should be able to handle mixed types of faces and not just triangles, half-edge data structures are less advisable.

To unfold a mesh, we are using a tree structure over the dual graph of the input mesh, which we call *unfold-tree* (see Section 2.1). Each tree-node stores the following values to make navigation and unfolding easy:

- A reference to the face it represents
- A link to the parent node
- A list of child nodes
- A transformation

The mentioned transformation rotates the represented face into the plane of the parent face along their shared edge. In other words, the transformation of a node unfolds the mesh by one step. The root node of the tree would contain an empty parent-link and the identity transformation. Unfolding the whole mesh is then done by a tree-traversal, where the transformations are applied successively. That way, unfolding has a time complexity of $\mathcal{O}(|F|)$ at maximum, with $|F|$ being the number of faces in the mesh.

To efficiently find nodes in the tree, we store each node in a list. The face-index is then equal to the index within that list. In a naive tree-implementation, searching a tree scales linear with the number of entries. By storing each node in a list, we can find every tree-node in constant time instead.

The resulting faces are stored in a 2D face representation. This data structure contains:

- A reference to the face it represents
- A list of 2D vertices
- A transformation

The transformation is the product of all small transformations from the unfold-tree. That way, each unfolded face contains information about how to transform the 3D face into 2D in one step. This information is beneficial, since it is often necessary to unfold just a small part of the whole mesh. When traversing only a subtree of the unfold-tree, we do not need to calculate the needed transformation for the subtree anew.

4.9 Pseudocode

All in all, our algorithm is summarized by the pseudocode of Algorithm 3.

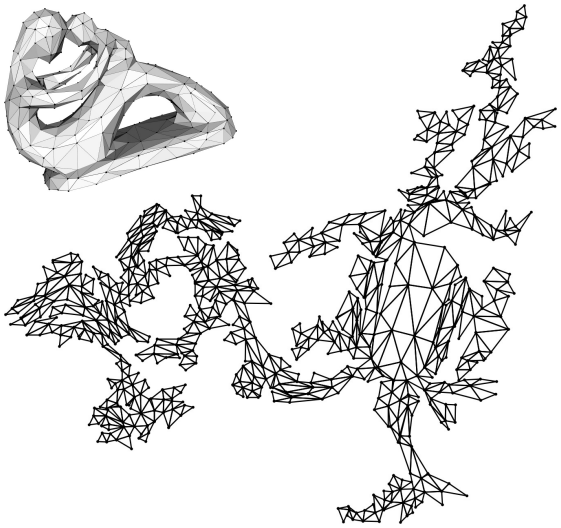
Algorithm 3 Tabu Search for Unfolding Polyhedra

```
Require: f                          ▷ f is the overlap detecting function
Require: x⃗                          ▷ x⃗ is the initial unfold-tree
  function TabuUnfolding(f, x⃗)
    while f(x⃗) > 0 do
      if stuckInRootLock() then                    ▷ See Section 4.5
        rerootIntoStuckSubTree(x⃗)
      else
        randomReroot(x⃗)
      end if
      if stuckInMemoryLock() then                  ▷ See Section 4.4
        clearMemory()
      end if
      b_H, x_H ← initHistoryValues()
      x ← selectRandomCollidingFace(x⃗)
      while x ≠ root(x⃗) do
        b = bestNeighbor(x)                        ▷ Filtered by tabu list
        if f(move(x⃗, x, b)) < f(x⃗) then
          x⃗ ← move(x⃗, x, b)
          memorize(x, b)
          continueOuterLoop
        end if
        if f(move(x⃗, x, b)) < f(move(x⃗, x_H, b_H)) then
                                   ▷ Better history value found
          b_H ← b
          x_H ← x
        end if
        x ← x.parent
      end while
            ▷ No improving solution found within the loop
      x⃗ ← move(x⃗, x_H, b_H)
      memorize(x_H, b_H)
    end while
  end function
```

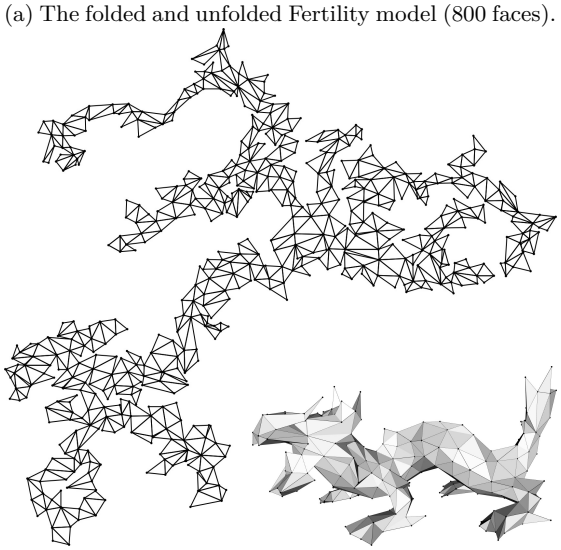
5 Results

An illustrative display of the whole unfolding pipeline for the Stanford Bunny can be found in Figure 12 in Appendix A. Other unfolding examples are shown in Figures 5a and 5b.

We compared the performance of our algorithm with two other methods (see Section 3): *Optimized Topological Surgery for Unfolding 3D Meshes* [26] and *Simulated Annealing to Unfold 3D Meshes and Assign Glue Tabs* [14]. We will refer to these methods as *OTS* and *SA*. For both methods an implementation was provided in the supplemental materials of the respective articles. To ensure fairness in our comparison, we removed the glue tab addition from the SA method, reducing it to an unfolding algorithm using simulated annealing. This is done to ensure comparability, since we do not add any glue tabs, which is a considerable overhead in computational time. Moreover, in the same implementation, we exchanged the naive overlap detection of the original implementation with a sweepline algorithm (see Section 4.7). This way, the overlap detection in all three methods works comparably fast, allowing for a better comparison between them. All three algorithms were implemented in C++ and compiled with the same compiler, using the same compilation flags.



(a) The folded and unfolded Fertility model (800 faces).



(b) The folded and unfolded Dragon model (600 faces).

Fig. 5: Different unfolding results.

5.1 Performance and Iterations

To compare the performance we measured the times each algorithm needed to unfold a set of given meshes. As a test set, we chose the Thingi10k [30] dataset. We filtered out all meshes which were non-manifold, consisted of multiple components, or were unrepresentable with at least 100 faces, resulting in 2,800 meshes. Each mesh has been tested in five different resolutions (100, 200, 400, 600, and 800 faces). The performances are plotted in Figure 6, and the success rates are plotted in Figure 7. More detailed results are shown in Table 2 in Appendix A.

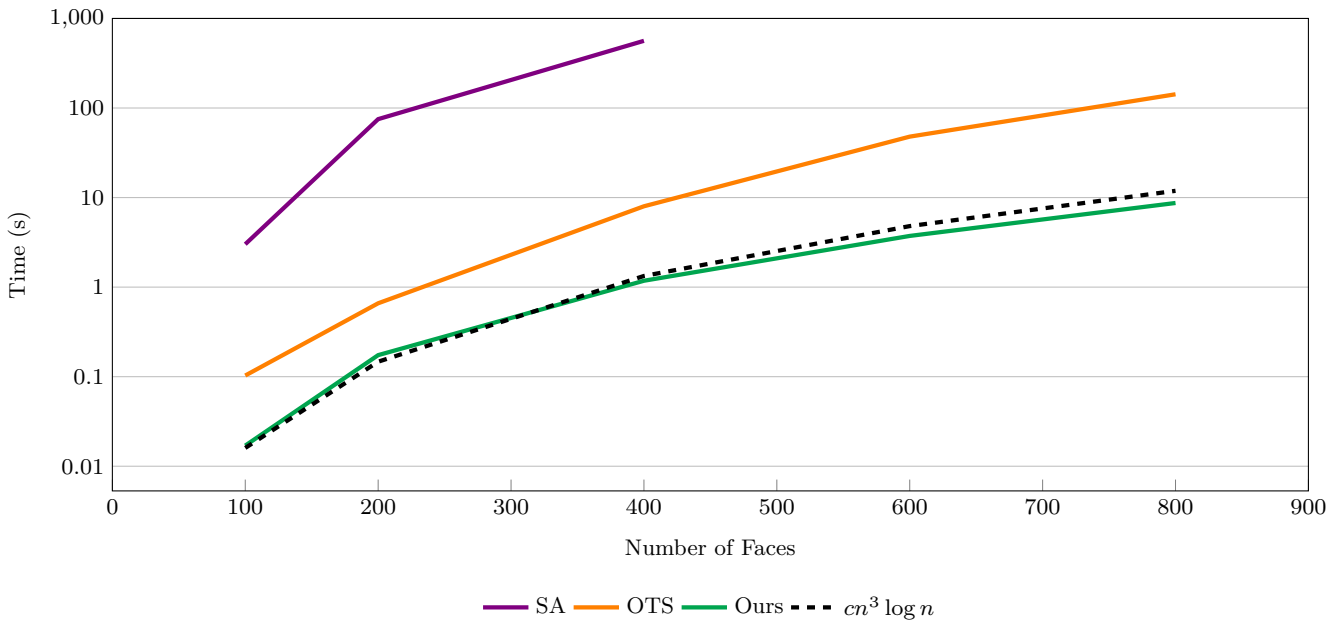


Fig. 6: Mean unfold timings of three methods. See Table 2 for exact values. To support our argumentation from Section 5.1.3, a polygon estimating the time complexity is plotted. The c represents a constant. Please note the log scale on the y-axis.

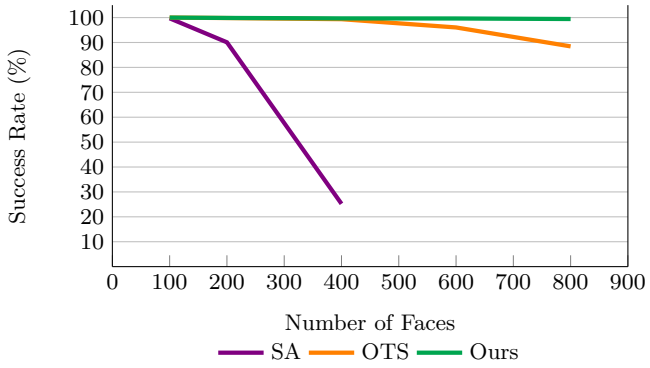


Fig. 7: Success rates for three methods. See Table 2 for exact values.

Our approach performs orders of magnitude faster than the other two, while being more reliable. For lower numbers of faces, the timings would permit to use our approach in interactive or even real-time applications.

5.1.1 Outlier Removal

To be able to determine a trend in our findings, we removed outliers per resolution. We define an outlier as any value greater than the mean plus three times the standard deviation of the underlying data. The timings presented in Figure 5 are filtered this way. For the unfiltered values, please refer to Table 2 in Appendix A.

5.1.2 Iterations

Apart from the performance, we also measured the number of iterations our algorithm needed to unfold the same dataset as in Section 5.1. The mean values of the filtered data (see Section 5.1.1) are shown in Figure 8. A clear linear relationship is visible. Therefore, we con-

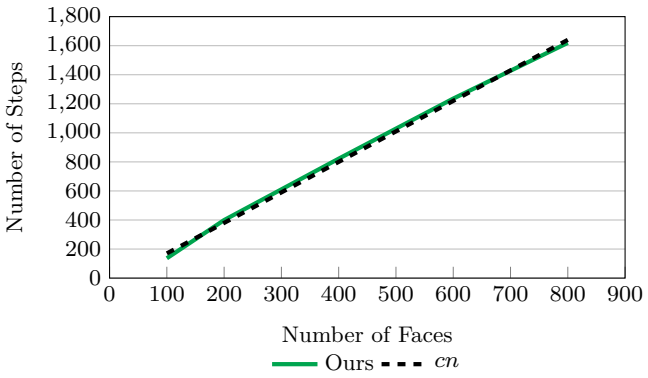


Fig. 8: Mean filtered iterations needed for our method. The c represents a constant. A clear linear relationship is visible.

clude that our algorithm on average needs $\mathcal{O}(|F)$ iterations to find an overlap-free unfolding for a triangular polyhedron with $|F|$ faces.

5.1.3 Complexity Estimation

The unfold-tree used in our algorithm in general has a non-constant branching factor. Within this section, we will refer to the branching factor of the ith node as b_i. For triangular meshes the branching factor is 2 in every node. On average, the branching factor of an unfold-tree is the average valence of the dual-graph it spans minus one $(val-1)$. In each iteration, our method performs an overlap detection for each possible move a face can do. For the ith face there is a maximum of b_i possible moves if it is a leaf node in the unfold-tree and $b_i - 1$ for inner nodes. Testing overlaps has a time-complexity of $\mathcal{O}(n)$ for leaf nodes, and $\mathcal{O}(n \log(n))$ for inner nodes on average. Since the time complexity for internal nodes is asymptotically worse, we will assume every test to be within this complexity class. Within each iteration, our algorithm may climb up the tree and test every node on its way to root. In the best case, each branch of a tree has a height of $\log n$. In the worst case, each branch has a height of n. On average, we assume a tree to have a height of $\frac{n+\log n}{2}$. Thus, our time complexity is estimated as $\mathcal{O}(n^2 \log n)$ per iteration on average. Combining this value with the findings from Section 5.1.2, the estimated average time complexity is $\mathcal{O}(n^3 \log n)$

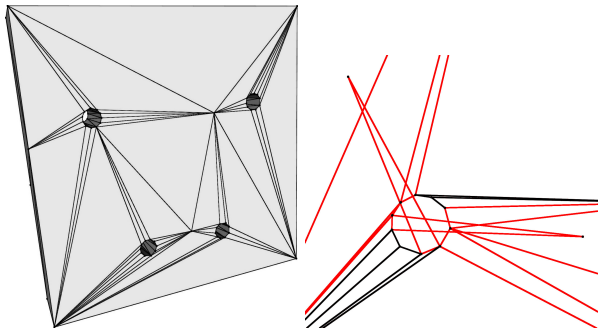
This estimation overshoots the measured filtered timings by a bit (see Figure 6). In our derivation, if we had to pick we always assumed the case that was worse, which was more conservative than it had to be. For example, it is possible that the overlap detection worked in $\mathcal{O}(n)$ instead of $\mathcal{O}(n \log n)$ in the vast majority of cases, which would erase a log term from the final result. This could have happened, if the majority of overlaps were located close to the leaves of the unfold-tree. Since on average half of the nodes of a tree are leaf nodes, this assumption is reasonable. We still conclude that our algorithm has a time complexity of

$$\mathcal{O}(n^3 \log n)$$

on average.

5.2 Failed Cases

In this section, we would like to highlight one case, where the approach by Takahashi et al. [26] found a net and our approach did not. The polyhedron is shown in Figure 9a. Unfolding this shape is particularly difficult, due to the coarse triangulation of the flat areas in combination with the holes, which are longer than they are wide. The latter property makes it impossible to unfold the walls of the tubes into the space defined by the opening of the tubes (see Figure 9b).



(a) A cuboid with holes, which our algorithm failed to unfold.

(b) A zoom-in on one of the holes in the unfolding. Red marks overlapping faces.

Fig. 9: Left: A polyhedron our approach failed to unfold in our test. Right: The problem posed by the holes in the cuboid.

A solution to this problem is to combine all triangles of the tubes into a triangle strip. The cuboid then has to be unfolded in a way that these strips can be placed at the outside. This solution is visualized in Figure 10.

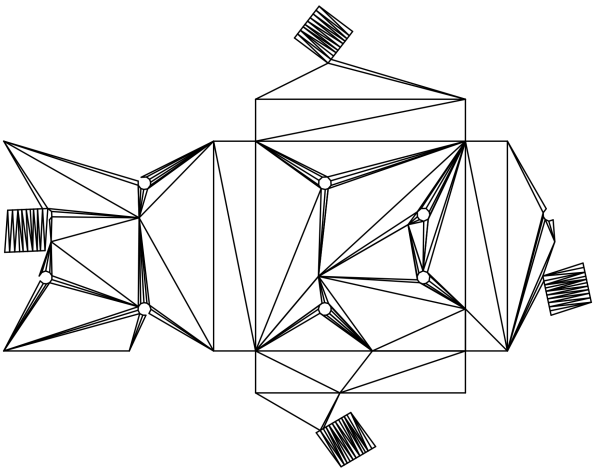
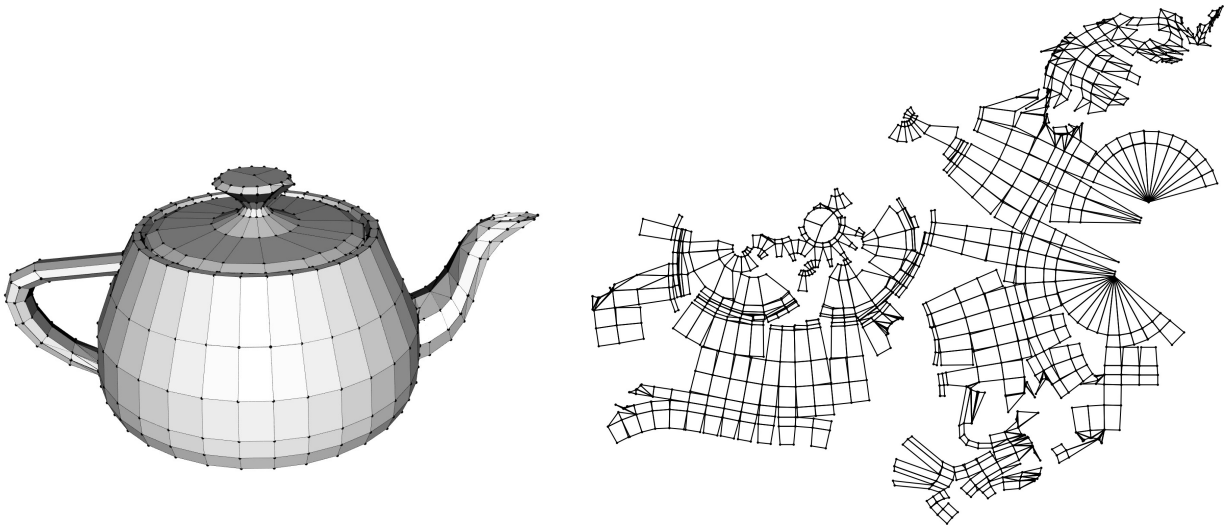


Fig. 10: A net for the polyhedron shown in Figure 9.

Finding such a solution is highly unlikely for a randomized algorithm without any geometric awareness, like ours.

Aside from geometrically extreme cases, like the one shown in this section, our algorithm is more reliable than the other two, as shown in Figure 7 and Table 2.



(a) A folded Utah Teapot with many non-triangular faces.

(b) The corresponding unfolding.

Fig. 11: A folded and unfolded Utah Teapot with 890 faces of different types.

5.3 Non-triangular Input

Besides performing way faster than the other two methods, our approach is also independent of the face type. While the other two methods in their current implementation are bound to triangles, we can process arbitrary face types, which can even be mixed. Figure 11 shows an unfolded Utah Teapot with different types of faces.

6 Conclusion and Future Work

In this work we presented an algorithm, which edge-unfolds a given mesh into a single-patched unfolding using tabu search. As input, our algorithm can process meshes with planar faces of arbitrary type. The proposed algorithm outperforms every known comparable algorithm by orders of magnitude. This improvement of speed and the nature of the algorithm permit interactive work with unfoldings of a few hundred faces. To this day, this has not been possible.

The notable limitation of the proposed algorithm is its inability to handle input meshes which are not-unfoldable. It is up to future research to extend or modify the given algorithm to detect and overcome not-unfoldability.

7 Acknowledgements

I would like to thank Prof. Dr. Marc Alexa for supervising my Master Thesis, which was the first step towards this publication. Moreover, I would like to thank Prof. Dr. Renato Pajarola for his support and very helpful discussions about local minima, as well as for supervising my PhD.

References

1. Alexa, M., Wardetzky, M.: Discrete laplacians on general polygonal meshes. ACM Transaction on Graphics **30**(4), 102:1–10 (2011). DOI 10.1145/2010324.1964997
2. An, B., Miyashita, S., Tolley, M.T., Aukes, D.M., Meeker, L., Demaine, E.D., Demaine, M.L., Wood, R.J., Rus, D.: An end-to-end approach to self-folding origami structures. IEEE Transactions on Robotics **34**(6), 1466–1473 (2018). DOI 10.1109/TRO.2018.2862882
3. Damian, M., Demaine, E.D., Flatland, R., O'Rourke, J.: Unfolding genus-2 orthogonal polyhedra with linear refinement. Graphs and Combinatorics **33**(5), 1357–1379 (2017). DOI 10.1007/s00373-017-1849-5
4. Damian, M., Flatland, R., O'Rourke, J.: Epsilon-unfolding orthogonal polyhedra. Graphs and Combinatorics **23**(1), 179–194 (2007). DOI 10.1007/s00373-007-0701-8
5. Demaine, E.D., Demaine, M.L., Eppstein, D., O'Rourke, J.: Some polycubes have no edge zipper unfolding. In: Proceedings Canadian Conference in Computational Geometry, pp. 101–105. Saskatchewan, Saskatoon, Canada (2020)
6. Demaine, E.D., O'Rourke, J.: Geometric Folding Algorithms: Linkages, Origami, Polyhedra. Cambridge University Press (2007). DOI 10.1017/CBO9780511735172
7. Demaine, E.D., Tachi, T.: Origamizer: A practical algorithm for folding any polyhedron. In: International Symposium on Computational Geometry, vol. 77, pp. 34:1–16. Schloss Dagstuhl – Leibniz-Zentrum für Informatik (2017). DOI 10.4230/LIPIcs.SoCG.2017.34
8. Dürer, A.: Underweysung Der Messung Mit Dem Zirkel Und Richtscheyt. Hieronymus Andreae, Nüremberg (1525)
9. Felton, S.M., Tolley, M.T., Shin, B., Onal, C.D., Demaine, E.D., Rus, D., Wood, R.J.: Self-folding with shape memory composites. Soft Matter **9**(32), 7688–7694 (2013). DOI 10.1039/C3SM51003D

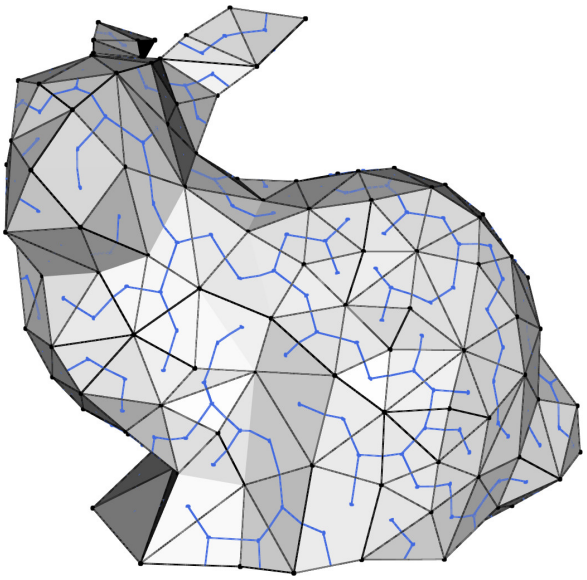
10. Glover, F.: Future paths for integer programming and links to artificial intelligence. Computers & Operations Research **13**(5), 533–549 (1986). DOI 10.1016/0305-0548(86)90048-1
11. Haenselmann, T., Effelsberg, W.: Optimal strategies for creating paper models from 3d objects. Multimedia Systems **18**(6), 519–532 (2012). DOI 10.1007/s00530-012-0273-1
12. Hao, Y., Kim, Y., Xi, Z., Lien, J.M.: Creating foldable polyhedral nets using evolution control. In: Robotics: Science and Systems, vol. 14, pp. 7:1–9 (2018). DOI 10.15607/RSS.2018.XIV.007
13. Ion, A., Rabinovich, M., Herholz, P., Sorkine-Hornung, O.: Shape approximation by developable wrapping. ACM Transactions on Graphics **39**(6), 200:1–12 (2020). DOI 10.1145/3414685.3417835
14. Korpitsch, T., Takahashi, S., Gröller, E., Wu, H.Y.: Simulated annealing to unfold 3d meshes and assign glue tabs. Journal of WSCG **28**(1-2), 47–56 (2020). DOI 10.24132/JWSCG.2020.28.6
15. Lang, R.J.: The Complete Book of Origami: Step-By-Step Instructions In Over 1000 Diagrams: 37 Original Models. Dover Origami Papercraft. Dover Publications (1988)
16. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Operations Research **21**(2), 498–516 (1973). DOI 10.1287/opre.21.2.498
17. Lu, B., Li, D., Tian, X.: Development trends in additive manufacturing and 3d printing. Engineering **1**(1), 85–89 (2015). DOI 10.15302/J-ENG-2015012
18. Mitani, J., Suzuki, H.: Making papercraft toys from meshes using strip-based approximate unfolding. ACM Transactions on Graphics **23**(3), 259–263 (2004). DOI 10.1145/1015706.1015711
19. Robinson, N.: The Origami Bible: A Practical Guide to The Art of Paper Folding. North Light Books (2004)
20. Rus, D., Tolley, M.T.: Design, fabrication and control of origami robots. Nature Reviews Materials **3**(6), 101–112 (2018). DOI 10.1038/s41578-018-0009-8
21. Schlickenrieder, W.: Nets of polyhedra. Diploma thesis, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin (1997)
22. Shatz, I., Tal, A., Leifman, G.: Paper craft models from meshes. The Visual Computer **22**(9), 825–834 (2006). DOI 10.1007/s00371-006-0067-6
23. Stein, O., Grinspun, E., Crane, K.: Developability of triangle meshes. ACM Transactions on Graphics **37**(4), 77:1–14 (2018). DOI 10.1145/3197517.3201303
24. Straub, R., Prautzsch, H.: Creating optimized cut-out sheets for paper models from meshes. Karlsruhe Reports in Informatics **36**, 1–15 (2011). DOI 10.5445/IR/1000025577
25. Tachi, T.: Origamizing polyhedral surfaces. IEEE Transactions on Visualization and Computer Graphics **16**(2), 298–311 (2009). DOI 10.1109/TVCG.2009.67
26. Takahashi, S., Wu, H.Y., Saw, S.H., Lin, C.C., Yen, H.C.: Optimized topological surgery for unfolding 3d meshes. Computer Graphics Forum **30**(7), 2077–2086 (2011). DOI 10.1111/j.1467-8659.2011.02053.x
27. Xi, Z., Kim, Y.H., Kim, Y.J., Lien, J.M.: Learning to segment and unfold polyhedral mesh from failures. Computers & Graphics **58**(C), 139–149 (2016). DOI 10.1016/j.cag.2016.05.022
28. Xi, Z., Lien, J.M.: Continuous unfolding of polyhedra - a motion planning approach. In: IEEE/RSJ International Conference on Intelligent Robots and Systems, pp. 3249–3254 (2015). DOI 10.1109/IROS.2015.7353828
29. Yao, M., Belke, C.H., Cui, H., Paik, J.: A reconfiguration strategy for modular robots using origami folding. International Journal of Robotics Research **38**(1), 73–89 (2019). DOI 10.1177/0278364918815757
30. Zhou, Q., Jacobson, A.: Thingi10k: A dataset of 10,000 3d-printing models. arXiv preprint arXiv:1605.04797 (2016). DOI 10.48550/arXiv.1605.04797

A Timings and Unfolding Results

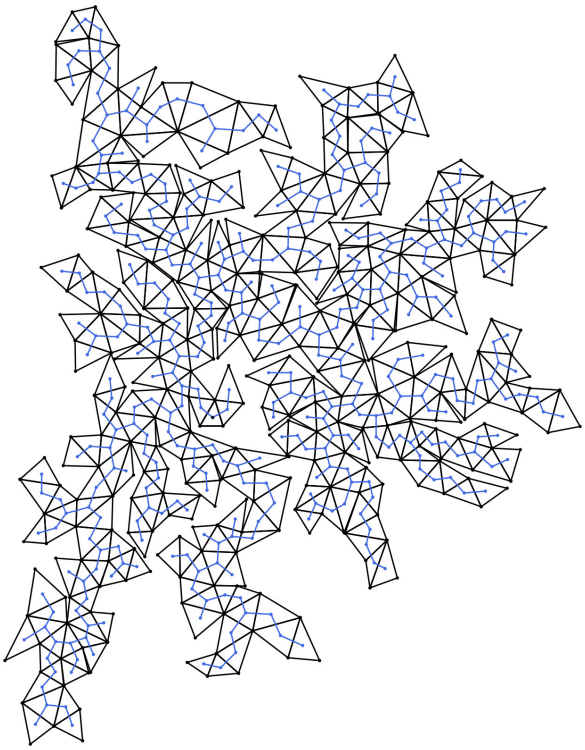
All timings of Table 2 were recorded on a Linux machine equipped with an i7-10700K CPU (3.8GHz) and 128GB RAM. A success was given, when a method was able to unfold the given mesh into a single-patched unfolding. For the OTS approach, a failure was given when the method yielded a segmented result. For the SA approach, a failure was given, when the method did not yield a result within a given number of iterations [14, Section 5]. Due to the low success-rate on 400 faces, the times of that row for the SA approach should be seen as an indicator, rather than a reliable number.

Value	|F|	OTS	SA	Ours
Min Time (s)	100	0.068	0.001	0.000
	200	0.190	0.114	0.000
	400	0.610	1.769	0.000
	600	1.966	-	0.000
	800	3.627	-	0.000
Mean Time (s)	100	0.107	3.674	0.021
	200	0.832	86.467	0.311
	400	10.923	565.600	1.890
	600	68.344	-	5.499
	800	211.713	-	13.769
Mean Time Filtered (s)	100	0.103	3.014	0.017
	200	0.660	75.106	0.174
	400	7.992	562.719	1.178
	600	47.895	-	3.729
	800	142.361	-	8.699
Max Time (s)	100	8.742	127.956	1.449
	200	270.273	612.973	102.130
	400	1,058.936	2,036.940	629.937
	600	3,847.608	-	1,017.305
	800	6,694.813	-	1,570.504
Success Rate (%)	100	100.00	99.71	100.00
	200	99.82	90.08	99.86
	400	99.36	25.23	99.68
	600	96.04	-	99.64
	800	88.44	-	99.43

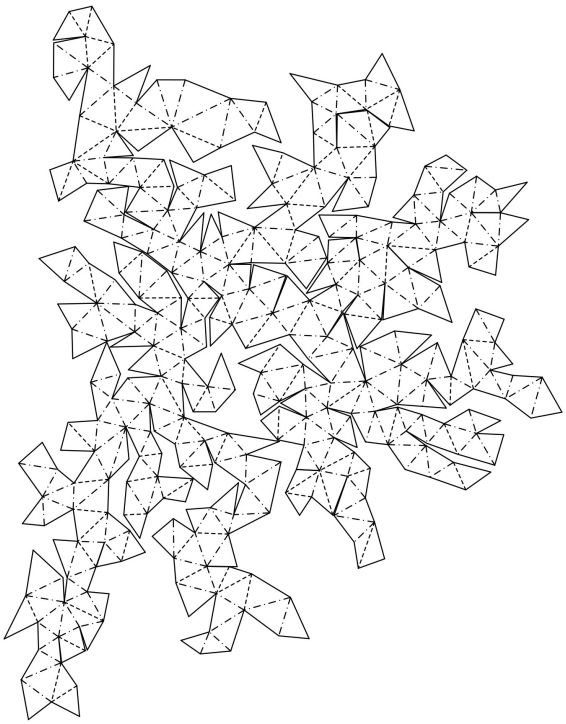
Table 2: Detailed timings, as well as success rates for the method of this article (*Ours*), as well as two similar approaches of the literature (*OTS* [26] and *SA* [14]).



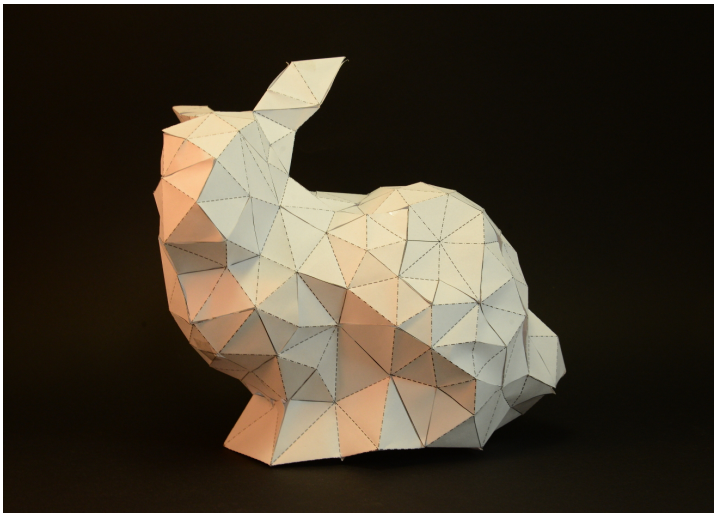
(a) The Stanford Bunny with 370 faces. The unfold-tree is visualized in blue.



(b) The unfolding of 12a. The unfold-tree is the same as in 12a.



(c) The unfold pattern of a simplified Stanford Bunny. Dashed lines represent valley folds and dash-dotted lines represent mountain folds.



(d) The manually refolded bunny.

Fig. 12: Unfolding the Stanford Bunny with 370 faces. Top left: The input mesh with the corresponding unfold tree. Top right: The calculated unfolding with the same unfold tree as in 12a. Bottom left: The unfold pattern, following the convention of dashed lines representing valley folds and dash-dotted lines representing mountain folds. Bottom right: The manually refolded bunny.