Unfolding polyhedra via tabu search

Lars Zawallich[1]

Abstract

Folding a discrete geometry from a flat sheet of material is one way to construct a 3D object. 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 sixth 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.

In recent years, the use of do-it-yourself decorations, arts, and toys crafted from unfoldings has become firmly established in commercial applications. Typically, these unfoldings are single-patched and have a low polygon count. Creating such a single-patched unfolding nowadays still needs a lot of time and effort. Improving the underlying techniques to create such a single-patched unfolding faster and more reliably than current approaches would allow for faster creation pipelines and finer polygonization in these applications.

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, cannot 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. Especially in manufacturing, single-patched unfoldings are preferable, since they allow for great efficiency [6, Section 22.1.1]. While 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.

✉ Lars Zawallich
  LarsZawallich@gmail.com

1 University of Zurich, Zurich, Switzerland

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

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

2 Background

Our algorithm is a tabu search algorithm [9], 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* [22]. 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.

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. A net can also be seen as a distortion-free parameterization with non-convex boundary.

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*.

There are different ways to look at an unfolding. One is to define it via its cutting of the graph (e.g., [25]). Another
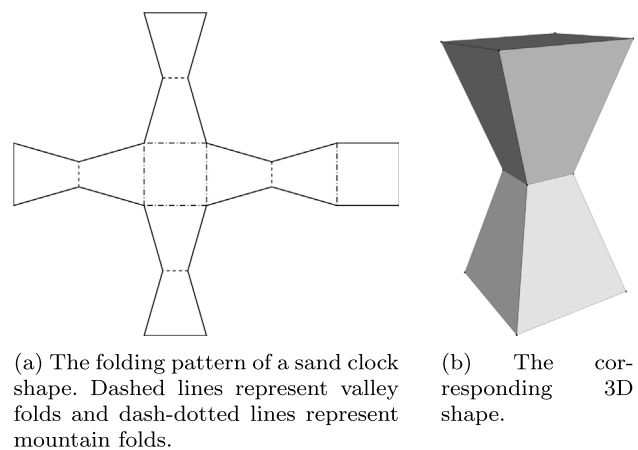
(a) The folding pattern of a sand clock shape. Dashed lines represent valley folds and dash-dotted lines represent mountain folds.

(b) The corresponding 3D shape.

Fig. 1 An example for line-styles representing different folding directions
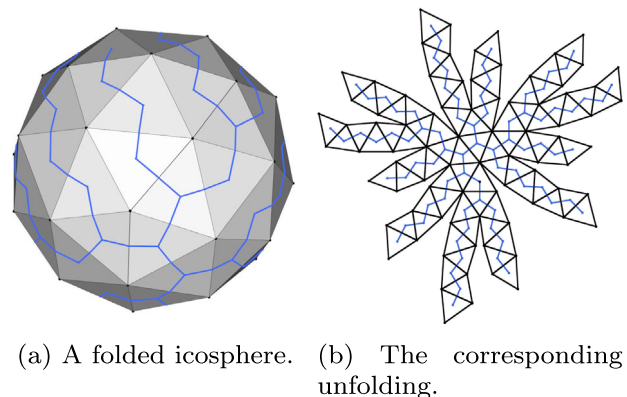
(a) A folded icosphere. (b) The corresponding unfolding.

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

way is to define the unfolding as a spanning-tree of the dual-graph of the mesh (e.g., [10]). 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*.

2.2 Tabu search

Tabu search is a metaheuristic for optimization and has first been introduced by Fred Glover [9]. The basic technique is outlined in Algorithm 1.

Algorithm 1 Tabu Search

function TABUSEARCH($\vec{x}$, $maxTabuSize$)
                                     $\triangleright$ $\vec{x}$ is an initial state of parameters
    tabuList = list()
    while !stoppingCriterion() **do**
        neighbors ← getNeighbors($\vec{x}$)
        neighbors ← removeTabus(neighbors, tabuList)
                                 $\triangleright$ Avoids undoing past steps
        bestNeighbor ← neighbors.best()
        tabuList.push($\vec{x}$)
        $\vec{x}$ ← bestNeighbor
        tabuList.shrinkToSize(maxTabuSize)
                                 $\triangleright$ 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) cannot 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 Sects. 4 and 4.4.

3 Related work

Besides the purpose of creating paper models or art, folding and unfolding polyhedra appear in other areas of research as well. In robotics, a recent publication suggested reconfiguring modular robots with folding and unfolding techniques [30]. 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].

The field of parameterization "only" differs from the field of unfolding by allowing distortions. Poranne et. al [18] optimized the cuts needed to parameterize. Sawhney and Crane [21] focused on usability without sacrificing quality. An overview of the field of parameterization has been given by Hormann et. al [12]. In contrast to parameterization, unfolding does not allow any distortion. Thus, any unfolding will also be a distortionless parameterization, with non-convex boundary.

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 [26]. 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 group of practical approaches for 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 [17], approximations with cones and cylinders [23], optimizing for hinges [24] 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. One major drawback of most segmentation approaches is the resulting segmentation being arbitrary in both the shape and the number of segments. This oftentimes leads to impractical segmentations, which are hard to reconnect. Our approach does not segment. Instead, it is applicable to user-defined segmentations, giving the user full control over the segmentation quality.

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 [25]. Their approach to remove overlaps is to cut the unfolding into several parts [25, 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 [10]. Takahashi et al. proposed to start off with small patches and stitch them together, using a genetic optimization algorithm [27]. 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 [28]. 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., [11, 29]) 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 [27]. The other approach uses simulated annealing to unfold a given mesh, while additionally considering gluetags [14]. In our experiments, this approach scales poorly and lacks reliability (see Figs. 8 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 (Sects. 4.1)
2. An initial unfolder (Sect. 4.2)
3. A strategy to select the best step (Sect. 4.3)
4. A strategy to overcome local minima (Sects. 4.4 and 4.5)
5. (Optional) Additional optimization parameters (Sect. 4.6)

Additionally, in Sect. 4.7 efficient overlap detection, in Sect. 4.8 the tabu mechanism, and in Sect. 4.9 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 [22]. 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.
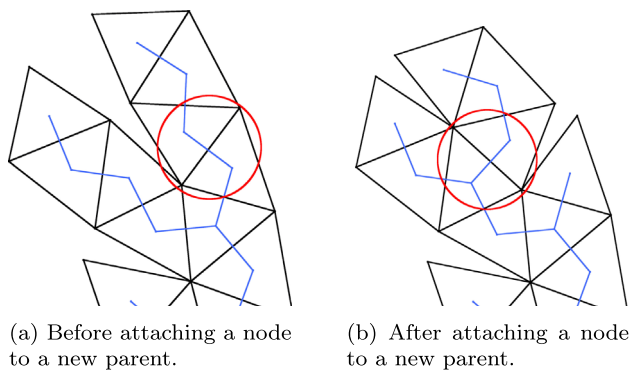
(a) Before attaching a node to a new parent.

(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

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 Fig. 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.

It may happen that the randomly selected face cannot 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 Fig. 4), or all possible dual-graph neighbors are in the tabu list (see Sect. 4.4). If a face cannot 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. In case the root node is reached, but no move would improve the overall number of overlaps, we perform the relative best move we found while climbing. Our experiments showed that searching a path between overlapping nodes can sometimes lead to a blocking situation. Therefore, we decided to search on a path to a random node of the tree. Since we reroot the tree in each iteration (see Sect. 4.5), it is convenient to pick the root node as this random node.

4.4 Local minima

Tabu search stores m past steps in a so-called tabu list. Any step in the tabu list cannot 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 cannot 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.
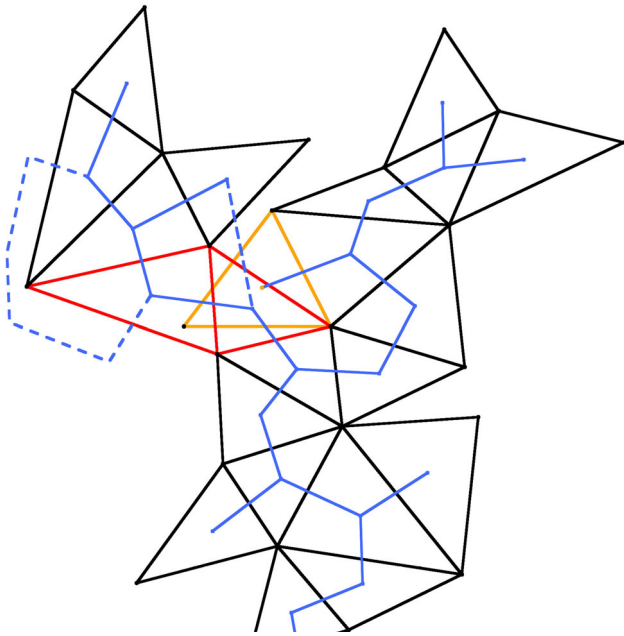
Fig. 4 A blocking situation where no colliding node or parent-node can be moved anymore. The root node is highlighted in orange, the unfold-tree is marked with blue lines, and overlapping faces are marked in red. In this case, the red faces are overlapping the root node. The root node is overlapping 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

4.5 Switching root nodes

In our data structure (Sect. 4.9), 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 cannot 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 Fig. 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 Fig. 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.

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 Sect. 4.3 needs to be extended as well, to allow selecting faces violating additional constraints. Lastly, the memory discussed in Sect. 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 cannot 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: $\vec{p}$ ▷ $\vec{p}$ is a list of unfolded polygons
  function DETECTOVERLAPS($\vec{p}$)
    $\vec{p}' \leftarrow sort(\vec{p}, \vec{d})$
            ▷ Sort by lower bounding box coordinate in dimension $\vec{d}$
    $overlaps \leftarrow 0$
    for $i \leftarrow 0$ **to** $|\vec{p}'|$ **do**
      $f_i \leftarrow \vec{p}'_i$
      for $j \leftarrow i + 1$ **to** $|\vec{p}'|$ **do**
        $f_j \leftarrow \vec{p}'_j$
        if $f_j.bBox.lower_{\vec{d}}() > f_i.bBox.upper_{\vec{d}}()$ **then**
          break
        end if
        if $f_i.intersects(f_j)$ **then**
          $overlaps \leftarrow overlaps + 1$
        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.

The polygon–polygon intersection test is done via line–line intersection tests. To prevent neighboring faces or flat regions from falsely being detected as overlaps, we add a small tolerance to each endpoint of a line in these tests.

4.8 Tabu mechanism

Whenever a face is moved, its index and the index of its former parent are memorized. If the number of memorized index-pairs exceeds the maximum size (see Sect. 4.4), the oldest items are removed. As long as an index-pair is stored in the tabu memory, the respective face is prevented from being connected to its former parent again. This implementation

is very robust and adds minimal overhead to the computational time. In our experiments, this approach showed better performance and equal reliability compared to any approach including the overall tree-structure.

4.9 Data structures

To unfold a mesh, we are using a tree structure over the dual graph of the input mesh, which we call *unfold-tree* (see Sect. 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.

We implemented the tabu mechanism by using two different data structures: a queue and a list of lists. The queue stores pairs of indices, which are not allowed to be connected. For each face, the list stores a list of indices which the face is not allowed to be connected to. This setup avoids iterating over the whole queue each time an element needs to be filtered, which would scale logarithmically with the number of faces in the mesh. Instead, each list entry can have, at most, as
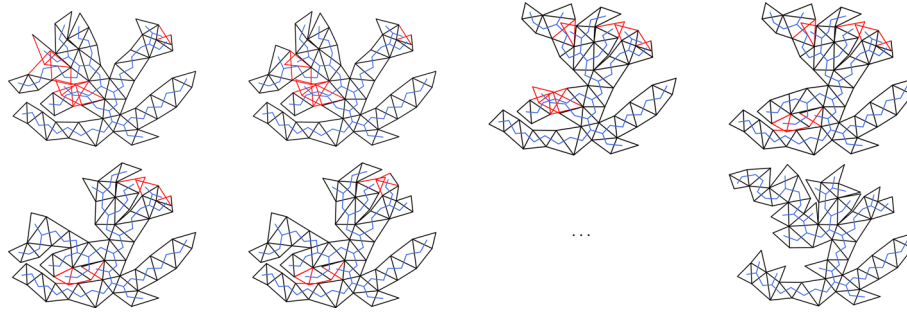
Fig. 5 Step-by-step visualization of our algorithm. Overlapping edges are marked in red. The unfold-tree is indicated with blue lines

many entries as there are edges on the respective face. This value depends only on the type of face and is independent of the number of faces.

4.10 Pseudocode

Our algorithm is summarized by the pseudocode in Algorithm 3. Moreover, Fig. 5 shows how our algorithm resolves overlaps in an unfolding step by step.

Algorithm 3 Tabu Search for Unfolding Polyhedra

Require: $\vec{x}$ ▷ $\vec{x}$ is the initial unfold-tree (see Sect. 4.2)
Require: f ▷ f is the overlap detecting function
  function TABUUNFOLDING($f, \vec{x}$)
    while $f(\vec{x}) > 0$ **do**
      if $stuckInRootLock()$ **then**   ▷ See Section 4.5
        $rerootIntoStuckSubTree(\vec{x})$
      else
        $randomReroot(\vec{x})$
      end if
      if $stuckInMemoryLock()$ **then**   ▷ See Section 4.4
        $clearMemory()$
      end if
      $b_H, x_H \leftarrow initHistoryValues()$
      $x \leftarrow selectRandomCollidingFace(\vec{x})$
      while $x \neq root(\vec{x})$ **do**
        $n = filteredNeighbors(x)$ ▷ See Section 4.8
        $b = bestNeighbor(n)$
        if $f(move(\vec{x}, x, b)) < f(\vec{x})$ **then**
          $\vec{x} \leftarrow move(\vec{x}, x, b)$
          $memorize(x, b)$
          continueOuterLoop
        end if
        if $f(move(\vec{x}, x, b)) < f(move(\vec{x}, x_H, b_H))$ **then**
             ▷ Better history value found
          $b_H \leftarrow b$
          $x_H \leftarrow x$
        end if
        $x \leftarrow x.parent$ ▷ See Section 4.3
      end while
         ▷ No improving solution found within the loop
      $\vec{x} \leftarrow move(\vec{x}, x_H, b_H)$
      $memorize(x_H, b_H)$
    end while
  end function

---

5 Results

An illustrative display of the whole unfolding pipeline for the Stanford Bunny can be found in Fig. 13 in Appendix A. Other unfolding examples are shown in Figs. 6a and 6b.

We compared the performance of our algorithm with two other methods (see Sect. 3): *Optimized Topological Surgery for Unfolding 3D Meshes* [27] 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 Sect. 4.7). This way, the overlap detection in all three methods works comparably fast, allowing for a better comparison between the actual approaches. All three algorithms were implemented in C++ and compiled with the same compiler, using the same compilation flags.

5.1 Outlier removal

To be able to determine a trend in our findings, we removed outliers. We define an outlier as any value greater than the mean plus three times the standard deviation of the underlying data. The timings and iterations presented in Figs. 8 and 9 are filtered this way. For the unfiltered values, please refer to Table 2 in Appendix A.

5.2 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 [31] 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

(a) The folded and unfolded Fertility model (800 faces).
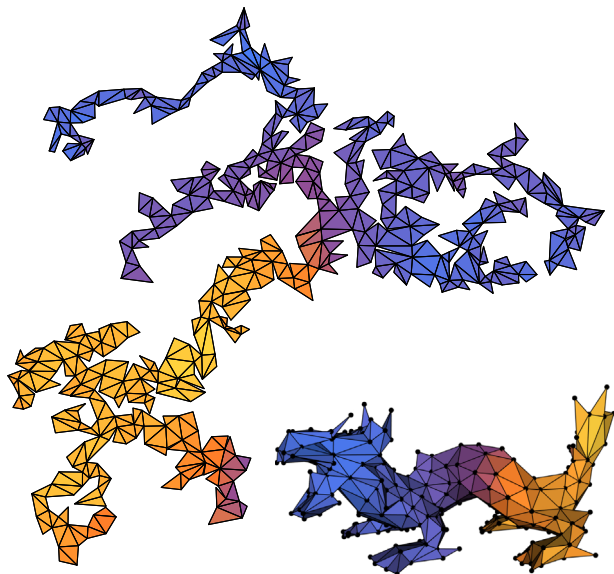
(b) The folded and unfolded Dragon model (600 faces).

Fig. 6 Different unfolding results

different resolutions (100, 200, 400, 600 and 800 faces), created by quadric edge collapses. The success rates are plotted in Fig. 7, and the performances are plotted in Fig. 8. More detailed results are shown in Table 2 in Appendix A.

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.

5.2.1 Iterations

Apart from the performance, we also measured the number of iterations our algorithm needed to unfold the same dataset as in Sect. 5.2. The mean values of the filtered data (see Sect. 5.1) are shown in Fig. 9.

A clear linear relationship is visible. Therefore, we conclude that our algorithm on average needs $\mathcal{O}(|F|)$ iterations to find an overlap-free unfolding for a triangular polyhedron with $|F|$ faces.

5.2.2 Complexity estimation

In each iteration, our method performs an overlap detection for each possible move a selected face can do. For any face the number of possible moves is determined by the number of edges it has, which is independent of the number of faces. Thus, we will assume this number to be a constant.

Even in a full tree the number of inner nodes and leaf nodes is almost balanced—there is one more leaf node than inner nodes. For every non-full tree this balance is shifted toward inner nodes. In the most extreme degenerate case (a list), there are $n - 1$ inner nodes and only one leaf node. Therefore, we will assume the number of inner nodes to be greater or equal to the number of leaf nodes on average.

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 we assumed to have at least as many inner nodes as leaf nodes and the time complexity for inner nodes is asymptotically worse, we will assume every overlap test to take $\mathcal{O}(n \log(n))$ time.

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 $\log n$ and in the worst case it has a height of n. To prevent underestimation, we will assume the tree to have a height of n on average.

Thus, our time complexity is estimated as $\mathcal{O}(n^2 \log n)$ per iteration on average. Combining this value with the findings
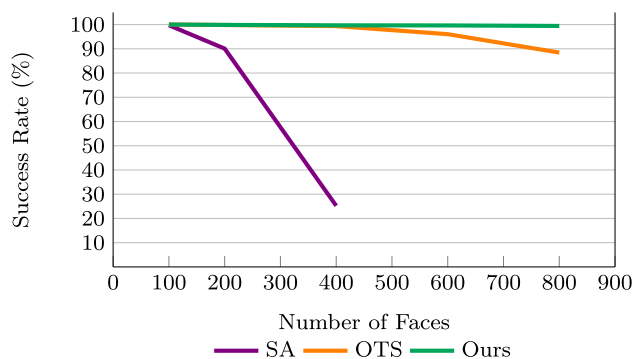

Fig. 7 Success rates for three methods. See Table 2 for exact values
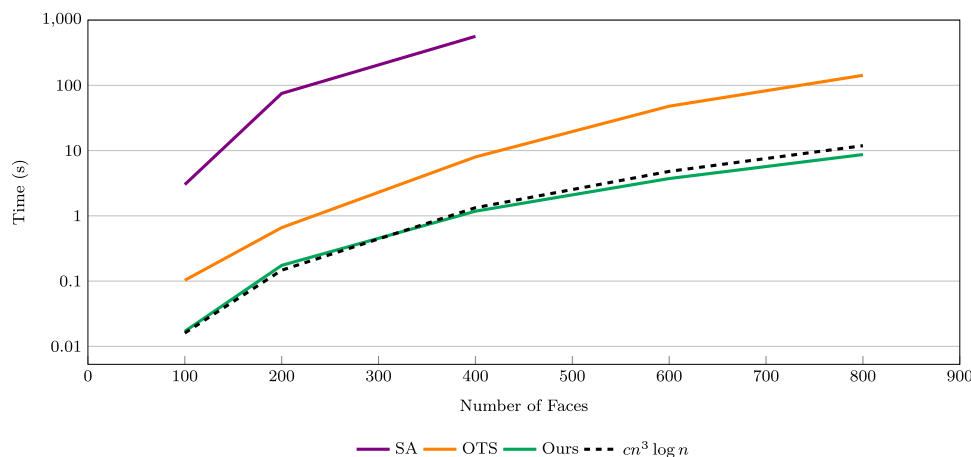
Fig. 8 Mean filtered unfold timings of three methods. See Table 2 for exact values. To support our argumentation from Sect. 5.2.2, a polygon estimating the time complexity is plotted. The c represents a constant. Please note the log scale on the y-axis
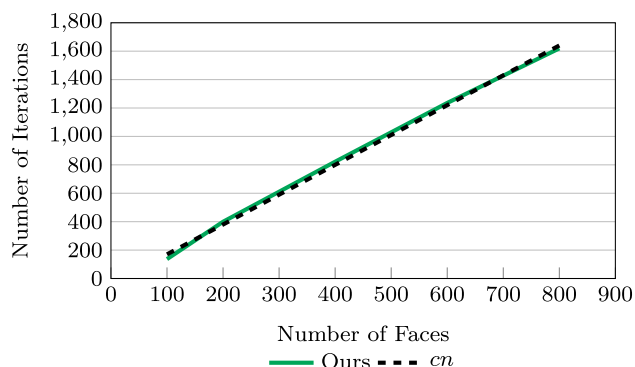

Fig. 9 Mean filtered iterations needed for our method. The c represents a constant. A clear linear relationship is visible

from Sect. 5.2.1, the estimated average time complexity is $\mathcal{O}(n^3 \log n)$.

This estimation overshoots the measured filtered timings by a bit (see Fig. 8). 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.

5.3 Failed cases

In this section, we would like to highlight one case, where the approach by Takahashi et al.[27] found a net and our approach did not. The polyhedron is shown in Fig. 10a.

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 Fig. 10b).

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 Fig. 11.

Finding such a solution is highly unlikely for a randomized algorithm without any geometric awareness, like ours. All models our algorithm was unable to unfold show similar features like the one of this section. In some areas they have spikes, dents or holes made from long triangles surrounded by an almost flat area with large triangles. We were able to manually unfold some of these models with a similar strategy as described above. Both comparing methods failed on most of these models as well. These are also the only models our algorithm failed to unfold, but comparing methods succeeded. Aside from geometrically extreme cases, like the one shown in this section, our algorithm is more reliable than the other two, as shown in Fig. 7 and Table 2.

(a) A cuboid with holes, which our algorithm failed to unfold.
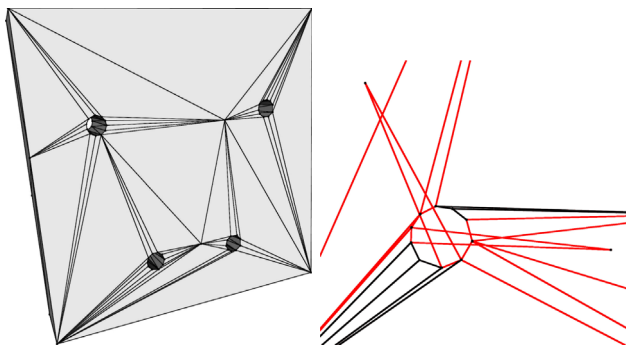
(b) A zoom-in on one of the holes in the unfolding. Red marks overlapping faces.

Fig. 10 Left: A polyhedron (Thingi-ID: 105860) our approach failed to unfold in our test. Right: The problem posed by the holes in the cuboid
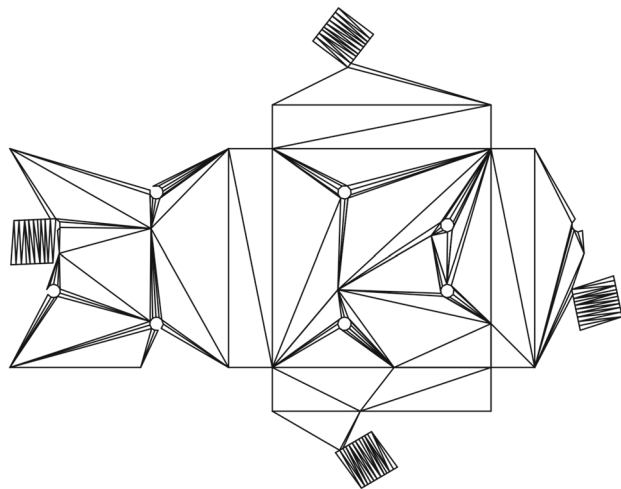

Fig. 11 A net for the polyhedron shown in Fig. 10

5.4 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 12 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.
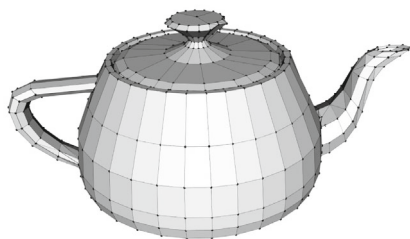
(a) A folded Utah Teapot with many non-triangular faces.
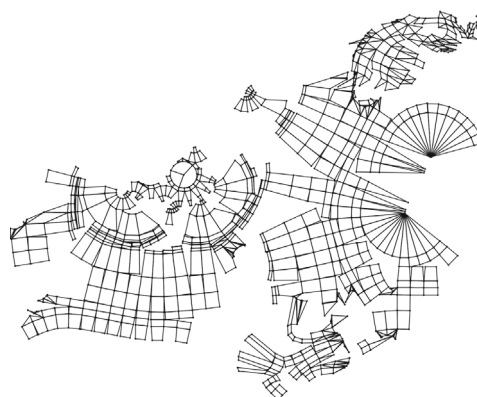
(b) The corresponding unfolding.

Fig. 12 A folded and unfolded Utah Teapot with 890 faces of different types

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.

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

(b) The unfolding of 13a. The unfold-tree is the same as in 13a.

(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.
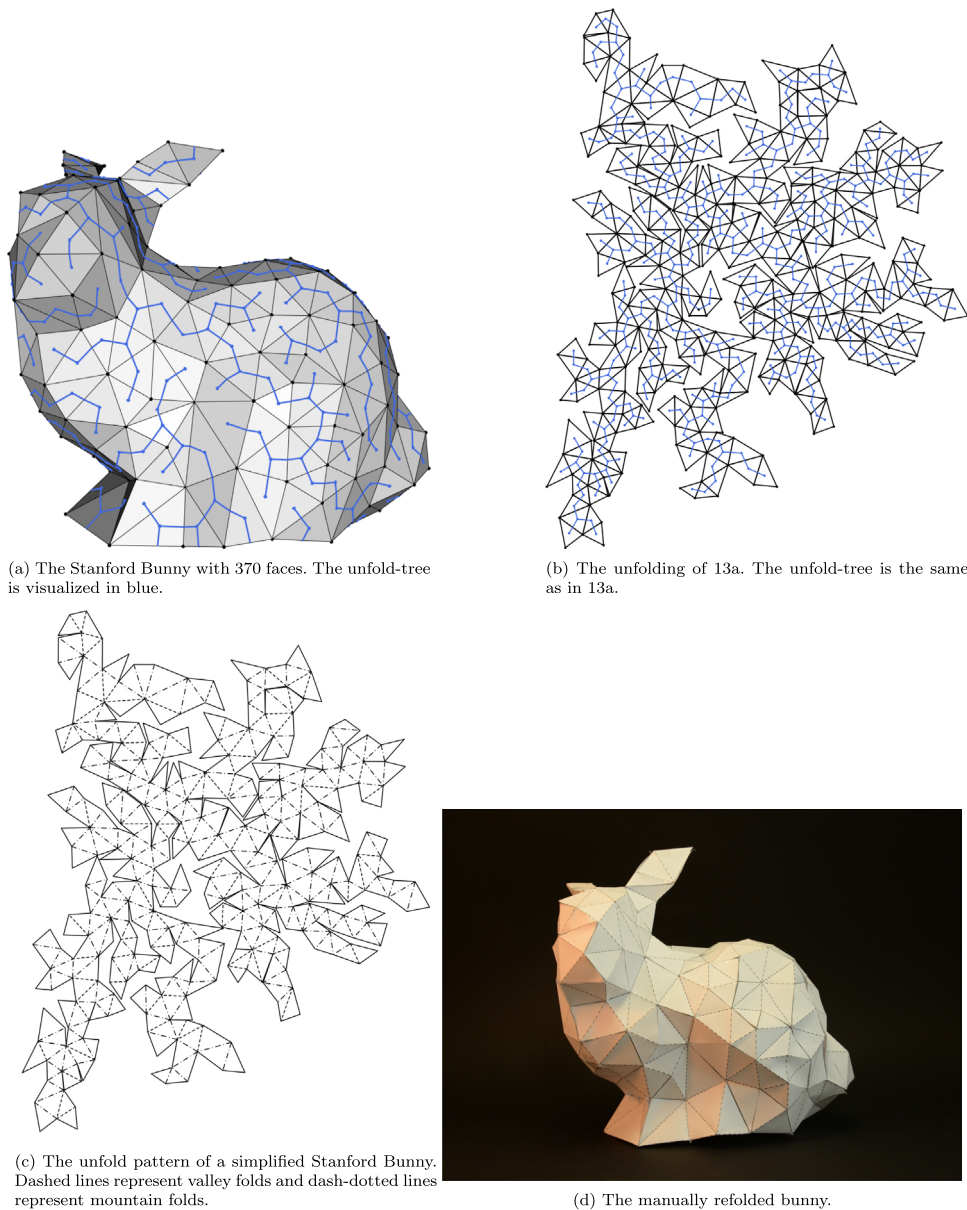
Fig. 13 Unfolding the Stanford Bunny with 370 faces. Top left: The input mesh with the corresponding unfold tree. Top right: The calculated unfolding with the same unfold tree as in 13a. Bottom left: The unfold pattern, following the convention of dashed lines representing valley folds and dash-dotted lines representing mountain folds. Bottom right: The manually refolded bunny

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* [27] and *SA* [14])

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

Declarations

Conflict of interest The authors declare that they have no Conflict of interest. This work involved no human participants or animals.

References

1. Alexa, M., Wardetzky, M.: Discrete laplacians on general polygonal meshes. ACM Transaction on Graphics **30**(4), 102:1–10 (2011). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/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). https://doi.org/10.4230/LIPIcs.SoCG.2017.34
8. Dürer, A.: Underweysung Der Messung Mit Dem Zirkel Und Richtscheyt. Hieronymus Andreae, Nüremberg (1525)
9. Glover, F.: Future paths for integer programming and links to artificial intelligence. Computers & Operations Research **13**(5), 533–549 (1986). https://doi.org/10.1016/0305-0548(86)90048-1
10. Haenselmann, T., Effelsberg, W.: Optimal strategies for creating paper models from 3d objects. Multimedia Systems **18**(6), 519–532 (2012). https://doi.org/10.1007/s00530-012-0273-1
11. Hao, Y., Kim, Y., Xi, Z., Lien, J.M.: Creating foldable polyhedral nets using evolution control. In: Robotics: Science and Systems **14**, 7:1-9 (2018)
12. Hormann, K., Lévy, B., Sheffer, A.: Mesh parameterization: Theory and practice. In: ACM SIGGRAPH 2007 Courses, p. 1-es. Association for Computing Machinery (2007). https://doi.org/10.1145/1281500.1281510
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). https://doi.org/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). https://doi.org/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 Publications, Dover Origami Papercraft (1988)
16. Lin, S., Kernighan, B.W.: An effective heuristic algorithm for the traveling-salesman problem. Operations Research **21**(2), 498–516 (1973). https://doi.org/10.1287/opre.21.2.498

17. Mitani, J., Suzuki, H.: Making papercraft toys from meshes using strip-based approximate unfolding. ACM Transactions on Graphics **23**(3), 259–263 (2004). https://doi.org/10.1145/1015706.1015711

18. Poranne, R., Tarini, M., Huber, S., Panozzo, D., Sorkine-Hornung, O.: Autocuts: Simultaneous distortion and cut optimization for uv mapping. ACM Transactions on Graphics **36**(6) (2017). https://doi.org/10.1145/3130800.3130845

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). https://doi.org/10.1038/s41578-018-0009-8

21. Sawhney, R., Crane, K.: Boundary first flattening. ACM Transactions on Graphics **37**(1), 5:1-5:14 (2017). https://doi.org/10.1145/3132705

22. Schlickenrieder, W.: Nets of polyhedra. Diploma thesis, Technische Universität Berlin, Straße des 17. Juni 135, 10623 Berlin (1997)

23. Shatz, I., Tal, A., Leifman, G.: Paper craft models from meshes. The Visual Computer **22**(9), 825–834 (2006). https://doi.org/10.1007/s00371-006-0067-6

24. Stein, O., Grinspun, E., Crane, K.: Developability of triangle meshes. ACM Transactions on Graphics **37**(4), 77:1–14 (2018). https://doi.org/10.1145/3197517.3201303

25. Straub, R., Prautzsch, H.: Creating optimized cut-out sheets for paper models from meshes. Karlsruhe Reports in Informatics **36**, 1–15 (2011). https://doi.org/10.5445/IR/1000025577

26. Tachi, T.: Origamizing polyhedral surfaces. IEEE Transactions on Visualization and Computer Graphics **16**(2), 298–311 (2009). https://doi.org/10.1109/TVCG.2009.67

27. 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). https://doi.org/10.1111/j.1467-8659.2011.02053.x

28. Z, Xi, 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). https://doi.org/10.1016/j.cag.2016.05.022

29. 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). https://doi.org/10.1109/IROS.2015.7353828

30. 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). https://doi.org/10.1177/0278364918815757

31. Zhou, Q., Jacobson, A.: Thingi10k: A dataset of 10,000 3d-printing models. arXiv preprint arXiv:1605.04797 (2016). https://doi.org/10.48550/arXiv.1605.04797
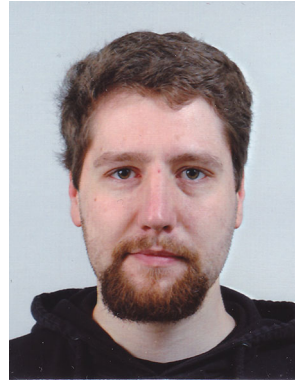
Lars Zawallich recieved his B.Sc. and M.Sc. in Computer Science at the Technical University of Berlin, in 2013 and 2016, respectively. He is currently a PhD candidate at the computer science department of the University of Zurich. His research interests include among others computer graphics, especially geometry processing, and optimization.