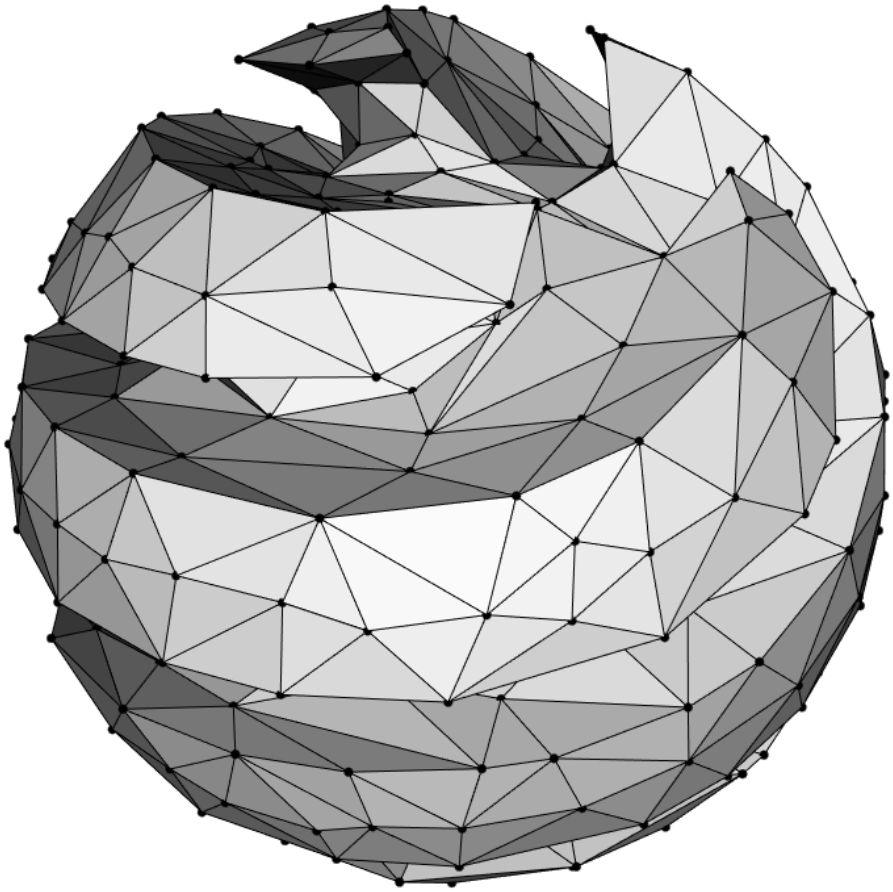
# Optimization Techniques in Unfolding



Bachelor's Thesis
23.08.2023

by Yves Meister, 19-703-255

Supervisors:
Prof. Dr. Renato Pajarola
Lars Zawallich

Visualization and MultiMedia Lab
Department of Informatics
University of Zürich

University of Zurich[UZH]

VISUALIZATIONANDMULTIMEDIALAB

# Abstract

This thesis presents an in-depth exploration of optimization algorithms aimed at addressing the challenging problem of unfolding 3D meshes by removing overlaps from initial unfoldings. Four distinct algorithms were selected for investigation: iterated local search (ILS), stochastic hill climbing (SHC), adaptive step size random search (ASSRS), and adaptive stochastic hill climbing (ASHC). Through implementation and experimentation, the performance of each algorithm was analyzed across varying mesh sizes and complexities. In the course of investigation, it became apparent that ILS struggled to deliver effective and efficient solutions, primarily due to its simplistic approach. ASSRS, a promising concept, faced challenges in its execution, with significant fail rates and a dependence on basic local search strategies. SHC, incorporating randomness to overcome local optima, demonstrated solid performance with success rates exceeding 93% and competitive runtimes. Notably, ASHC emerged as the standout algorithm, enhancing SHC through adaptive probabilities of making unfavorable moves as overlap counts decrease. ASHC consistently outperformed the other algorithms, showcasing the potential of adaptiveness in computational unfolding. Comparison with related works revealed ASHC's competitive edge, outperforming simulated annealing and performing on par with a genetic algorithm. As a result, this thesis contributes valuable insights into the realm of 3D mesh unfolding optimization, paving the way for future refinements of ASHC and potential advancements in the unfolding of complex 3D structures.

# Contents

# 1 Introduction

Folding and gluing a two-dimensional representation of a cube back together into its three-dimensional counterpart, also called papercraft, is a task that most of us have done at least once in our lifetimes. Some of us may even have been so lucky as to have had complex papercraft templates like trains, animals or others to enjoy. This very concept of papercraft is the foundation of this thesis. The goal is to get from a 3D object, or more accurately its approximated 3D mesh, to a 2D representation composed entirely of planar polygons, that can then theoretically be reassembled into the original object's mesh by folding. As one can imagine, this task is quite simple on objects such as a cube, but gets increasingly more difficult with the complexity of the shape.

There exists another reason apart from having fun to invest time and effort into the research of optimized unfolding, as Haenselmann and Effelsberg also state [HE12, chapter 1]. For the casual consumer, creating real-life physical 3D objects that can relatively accurately display any object of reasonable complexity is easily possible using 3D printers. The issue is that not everyone has the funds to afford the expensive equipment. With papercraft, one could build prototypes or models of custom shapes with the low cost of a regular printer, paper, glue and a pair of scissors. Optimized unfolding could also be used for more industrial cases. With the use of CNC *(computer numerical controlled)* machines with plasma or laser cutters, as well as CNC milling or drilling machines more durable materials like metal and wood can be accurately cut and manipulated. This opens the possibility of folding with materials like aluminum, for example, to create more durable prototypes.

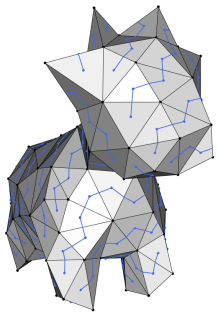


Figure 1.1: Approximation of 3D cow using 200 triangular faces

The way to unfold a three-dimensional object is to begin by approximating its surface using planar polygons as shown in Figure 1.1, creating a 3D mesh model. Exclusively triangles will be used in this thesis, but as any planar polygon can be split into triangles, an approach with triangles works for any number of polygon corners. The object mesh is then unfolded. This is achieved by converting the 3D mesh into a graph, then into a dual graph and then finding a spanning tree on said dual graph. This spanning tree, where each vertex is the face of a triangle, is then unraveled. After this process it will look like Figure 1.2.

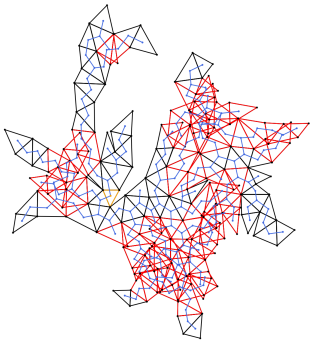


Figure 1.2: Unfolded cow with overlaps (shown in red)

The problem that remains is overlapping triangles. With complex objects these overlaps are almost unavoidable. Optimization techniques, such as simulated annealing [KTGW20] or genetic algorithms [TWS+11] can then be used to move around polygons in the 2D patch until a configuration is reached, where there are no more overlaps, as in Figure 1.3. The triangles may also not be warped in the process and must be joined together within a singular coherent collection of triangles at all times. Meaning, we cannot use the same approach as [TWS+11] where the triangles are split into many small groups and then stitched back together at a later time.

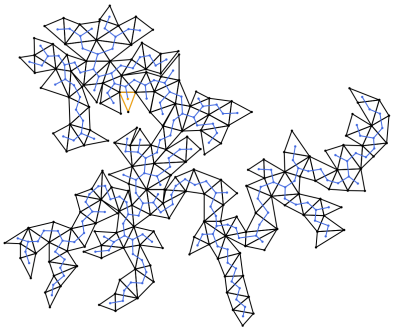


Figure 1.3: Unfolded cow after applying optimization techniques to remove overlaps

The goal of this thesis is to find and implement different optimization algorithms that can reach the aforementioned non-overlapping state quickly, ideally quicker than algorithms that have already been implemented to solve this problem. Secondly, a discussion and comparison of the effectiveness and efficiency of these algorithms shall be given in the hopes of optimizing the unfolding process further.

Similar problems have already been solved by Korpitsch et al. [KTGW20] and Takahashi et al. [TWS+11] using simulated annealing and genetic algorithms respectively. The main differences are that Korpitsch et al. added trapezoidal glue tabs to their 2D patches in order to be able to later glue them back together into the original 3D object for real-life applications, while Takahashi et al. unfolded their objects into multiple incoherent small patches of polygons and then used a genetic algorithm to stitch them back together overlap-free into a coherent patch.

In this thesis, the following optimization algorithms, sometimes slightly altered to better accommodate the problem at hand, will be used as stated in Section 4: Iterated local search, stochastic hill climbing, adaptive stochastic hill climbing, and adaptive step size random search.

The following chapters are organized as follows. In Section 2, an in-depth look at the fundamental and essential concepts underlying the problem will be given. Section 3 presents a thorough examination of the existing research in the field of unfolding, arranged in order of decreasing relevance to the subject of this thesis. This section offers valuable insights into related papers, highlights the similarities and differences with this thesis, and

delves into the algorithmic foundations of their respective approaches. In Section 4, the theoretical solution will be shown by introducing and explaining the algorithms used, showing their strengths and weaknesses. In Section 5, the practical, programmatic implementation of the before-discussed algorithms is shown and explained. Section 6 will unveil the gained results, while Section 7 engages in an analysis of these results and Section 8 draws a conclusion and discusses further work. For further insights into the algorithms, the appendix in Section 10 contains pseudocode, making it easier to grasp each algorithm.

# 2 Key Concepts

When discussing optimization techniques in unfolding, one must first gain an idea and overview of what both of those concepts entail. This section explains the basics of computational unfolding, as well as what optimization is and why it is necessary for the problem at hand. Then, it will be clarified why certain groups of optimization algorithms are impractical for unfolding. As most of the algorithms discussed are reliant on the concept of local search algorithms, a brief explanation of their inner workings is provided.

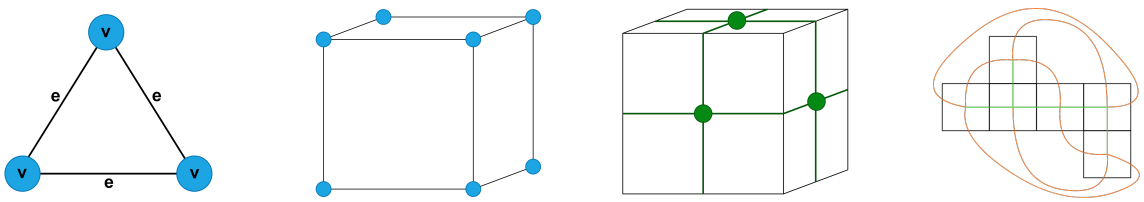
## 2.1 3D Meshes

3D meshes in computer graphics can be understood as simplified and approximated representations of more complex 3D objects. Such a mesh is constructed out of connected faces, edges and vertices, which form planar polygonal surfaces, to closely represent the original shape of the 3D object. Figure 1.1 shows an example of a 3D mesh.

## 2.2 Computational Unfolding

The field of computational unfolding is vast, containing a multitude of definitions, techniques and viewpoints. The following section will shed a light on a select number that are relevant or closely related to the approaches taken in this thesis.

The most basic concept of computational unfolding can be described as follows: "From a computer science perspective, unfolding a 3D model reduces a graph to a spanning tree." [HE12, chapter 3.2]. A graph $G$ contains a set of Vertices $V$ and a set of Edges $E$. Every edge $e \, \epsilon \, E$ is associated with two vertices $v \, \epsilon \, V$, as demonstrated in Figure 2.1(a) and every space enclosed by edges is called a face. Of our 3D mesh that should be unfolded, every edge is now treated as a graph edge while every corner is treated as a graph vertex. If we were trying to unfold a cube then Figure 2.1(b) shows this step. Next the corresponding dual graph is generated. A dual graph is a graph that uses the original graph's faces as its vertices and keeps the edges. The cube's dual graph is shown in green in Figure 2.1(c). By extension the vertices of $G$ turn into the faces of $D$. A spanning tree $T$ is then created from the dual graph. A spanning tree being a sub graph, where all vertices are connected by the minimum number of edges possible. The 3D mesh cube in the example Figure 2.1(d), is then cut along the edges that are not in the spanning tree: $e \, \epsilon \, E_D \backslash E_T$ (orange edges) and then folded open along the edges that are in the spanning tree: $e \, \epsilon \, E_T$ (light green edges).




(a) (b) (c) (d)

Figure 2.1: (a) Example of a graph $G$ composed of three blue vertices $v$ and three edges $e$. (b) 3D mesh of a cube transformed into graph $G$ composed of blue vertices and black edges. (c) Dual graph $D$ of graph $G$ with green vertices and green edges. (d) Unfolded version of the cube according to the underlying spanning tree. Green edges are part of the spanning tree while orange edges are not.

The issue that can arise is overlaps, as seen in Figure 1.2. If the spanning tree is chosen randomly on complex enough 3D meshes, they are almost guaranteed. An overlap is defined as two polygons of the original mesh occupying the same space in the 2D representation, shown as red triangles in Figure 1.2. To remove overlaps, polygons are moved around. Every polygon has a finite number of places to be moved to according to the dual graph $D$, as its corresponding dual graph vertex must stay connected to at least one other vertex via an edge. These moves are made until an overlap-free configuration is achieved. The number of possible moves at any given time is huge, and therefore, not all of them can be tried in a reasonable time frame. This problem can be solved using optimization.

## 2.3 Optimization

The following sections will explain some of the basics of optimization and why they are necessary for this thesis. During this section and going forward, whenever an algorithm is explained and can easily be shown on a graph spanning just two dimensions, this will be done with the help of visualization. For the visualization, the function, from now on referred to as 'base function', $f(x) = sin(x) + sin(\frac{5}{2}x),\ x\ \epsilon\ [-3, 9]$, shown in Figure 2.2, has been chosen as it contains some very prominent global and local optima.

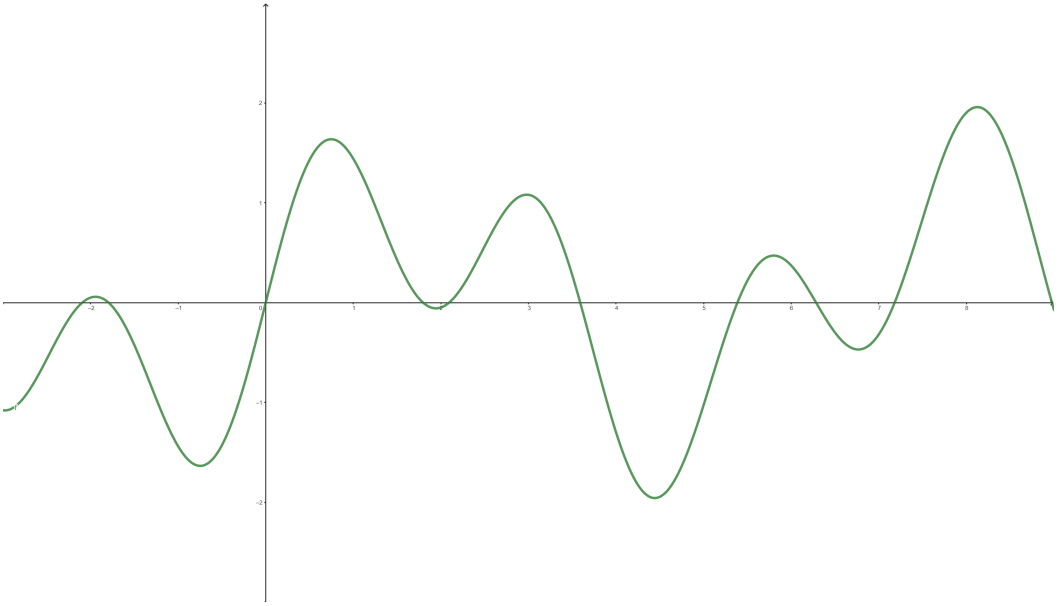


Figure 2.2: Base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x\ \epsilon\ [-3, 9]$ used for the visualization of following optimization algorithms

### 2.3.1 Basics of Optimization

The following short introduction to the basics of optimization is largely based on "An Introduction to Continuous Optimization"[AEP20, page 3-4] and "Optimization Theory"[JMT04, pages ix-xi]. Optimization is a field in mathematics, economics, engineering, and computer science that concerns itself with the search of optimal solutions to problems. These problems can oftentimes be defined by a function, and the task of the optimization algorithm is to find a global maximum or minimum over that function, given a certain set of constraints. The function, mostly called objective or cost function, is the quantity that is to be optimized, while the constraints show the limits of the available variables which define the problem.

As one can imagine, mathematical problems are extraordinarily diverse and often reflect real-life problems. This means that the resulting objective functions can differ greatly from one another. The result of this is that a plethora of optimization algorithms are necessary to solve this wide array of problems. A further factor is that of-

ten, optimization problems are quite complex. This leads to the question of whether you would like an algorithm to be primarily fast or accurate. Meaning, there are often algorithms designed to quickly solve a problem and find a good enough answer, and algorithms designed to find the perfect answer with a larger cost of time.

Generally speaking an optimization problem can be defined as follows (taken from [BV04, chapter 4.1.1]):

$$\begin{array}{ll} \text{minimize(or maximize)} & f(x) \\ \text{subject to} & f_i(x) \leq 0, \quad i = 1, ..., m \\ & h_i(x) = 0, \quad i = 1, ..., p \end{array}$$

Where:

- $f(x)$ is an objective function to be optimized.
- $f_i(x)$ are inequality constraints.
- $h_i(x)$ are equality constraints.

There are many different optimization algorithms, yet most of them operate in a similar way to achieve their main goal of finding optima. They iteratively search through their problem's solution space, navigating towards a minimum or maximum. The ways in which they search and evaluate potential solutions vary drastically. Most optimization algorithms, including those mentioned in this thesis, follow these rough steps to reach their goal:

First, they make an initial guess. The initial guess can be one or multiple potential solutions to the problem and is usually made randomly or based on some knowledge of the problem or search space.

Second, it must evaluate the current guess. This means checking what the current guess translates to on the objective function. In the case of this thesis, this step would involve evaluating how many overlaps a guess produces.

Third, the current guess is updated or replaced. In local search based algorithms, this is done by replacing the current guess with a neighboring state.

Fourth, it is checked whether a termination criterion has been reached. This can be if a solution is found that is good enough or after a set amount of iterations. Steps two to five are then repeated until a termination criterion is reached, either leaving us with a sufficient solution or an insufficient solution after the maximum runtime.

One of the best-known optimization approaches is the one used in algorithms such as Newton's method or gradient descent: Calculating the derivative of the objective function to figure out where to move iteratively. These methods are best used on optimization problems that operate in continuous space.

### 2.3.2 Continuous Optimization

Continuous optimization is the field of optimization where the variables can take any value in the real numbers [AEP20, page 12], as long as they adhere to any given constraints. Derivative or gradient based algorithms play a large part in continuous optimization. For example, with the gradient descent algorithm, the gradient or derivative

gives insight into the steepness of increases and decreases in the search space, allowing the algorithm to converge towards an optimal solution.

Apart from the basic and well-known algorithms such as Newton's method or gradient descent, many modern algorithms have been developed that mimic certain processes in nature. One example being genetic algorithms that imitate natural selection and evolution to create increasingly good solutions over many generations and can efficiently explore search spaces [HMR16, chapter 1]. Another example being swarm intelligence algorithms, such as particle swarm optimization or ant colony optimization, which mimic the collective behavior of insect swarms to cover the search space as further discussed in Section 2.4.2 [DBS06] [MW15].

### 2.3.3 Discrete Optimization

Discrete optimization is the field of optimization where the variables cannot take on all real values, but are limited to a predefined discrete set of values. Discrete optimization is often seen in combinatorial optimization problems. The very well-known traveling salesman problem belongs to this category [PR88, pages 1-3]. Since the search space in discrete problems is fundamentally different from the search space of continuous problems, many of the basic algorithms mentioned in Section 2.3.2 are not applicable. Especially gradient based algorithms can no longer be used, as derivatives cannot be calculated over the discrete search space.

This leads to the need for different types of algorithms to solve such problems. The most well-known and exclusively used in this thesis are metaheurisitc algorithms [GK03, pages xi-xii]. They combine the basic local search iteration loop and add ways to escape local optima or increase efficiency. Some examples are simulated annealing, tabu search, and iterated local search.

## 2.4 Impractical Algorithms

This section briefly explains why certain types of well-known algorithms are not well-suited to solve the problem of optimized unfolding.

### 2.4.1 Derivative Based Algorithms

Derivative based algorithms, such as gradient descent, are quite well-known and widely used. Not being able to use them to solve our problem of optimizing the unfolding process eliminates a big chunk of available algorithms.

Derivative based algorithms, as the name suggests, need to calculate a derivative in order to find an optimum. For example, gradient descent calculates derivatives between points in order to find the steepest ascent or descent, to iteratively move toward an optimal solution in the search space. In our problem, moving from one point in the search space to another corresponds to moving the faces of the mesh around. These moves are discrete, not continuous. The face is either completely moved somewhere or not at all. Consequently, a derivative over these basic movement operations cannot be calculated.

Obviously a derivative based approach can still be forced onto the problem. For instance, when unfolding a 3D mesh, one could assign weights to the edges of the dual graph $D$ before finding a minimum spanning tree $T$. Then, when making moves on our 2D collection of polygons, we can view movement not as discretely moving one triangle to a different position, but as continuously changing the weights on $D$, leading to changes in $T$. The remaining issues here are quite obvious, as the changes in our weights, while technically continuous, are upon closer inspection actually discrete. This stems from the fact that the weight change makes no difference to the mesh until it reaches a certain threshold, at which point some faces are moved due to the new minimum spanning tree. In essence, we are merely disguising our discrete operation as continuous.

One could still attempt this approach, but there is little to no promise that it would efficiently use the strengths of derivative based optimization algorithms.

### 2.4.2 Swarm Intelligence Algorithms

Swarm intelligence algorithms, such as particle swarm optimization (PSO) or ant colony optimization (ACO), may be less common and numerous than the derivative based ones, but are also not suitable for the purposes of this thesis. They are population-based optimization methods inspired by colonies of social insects, like bees or ants. While they are applicable for discrete problems, they are still impractical for our purpose. Swarm intelligence algorithms thrive on smooth and regular objective functions, which may also be discrete. However, the problem of mesh unfolding creates very irregular objective functions. The swarms, which operate by stochastically and collectively traversing through large search spaces, would most likely be inefficient in navigating these irregular functions.

## 2.5 Local Search Algorithms

Local search algorithms are a group of algorithms that all rely on the same fundamental concepts. Many of the algorithms discussed in this thesis, such as simulated annealing and iterated local search, are based on these concepts. The following explanation is based on the book "Theoretical Aspects of Local Search" [MKA07, definition 1.3 & definition 1.6].The pseudocode in Listing 10.2 demonstrates these concepts using the same variables as the following explanation.

### 2.5.1 Explanation of Local Search

Local search starts off by looking at a first potential solution $P$ as shown in Figure 2.3.

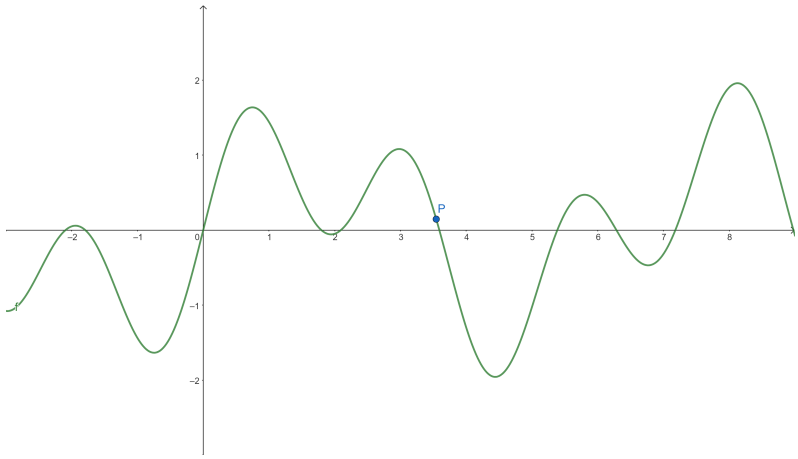



Figure 2.3: Visualization of the starting state of a local search algorithm over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x\ \epsilon\ [-3, 9]$.

It then enters a loop, where it iteratively looks at neighboring solutions $N$ and checks if the best neighbor $bN$ is better than the current potential solution $P$ as shown in Figure 2.4.

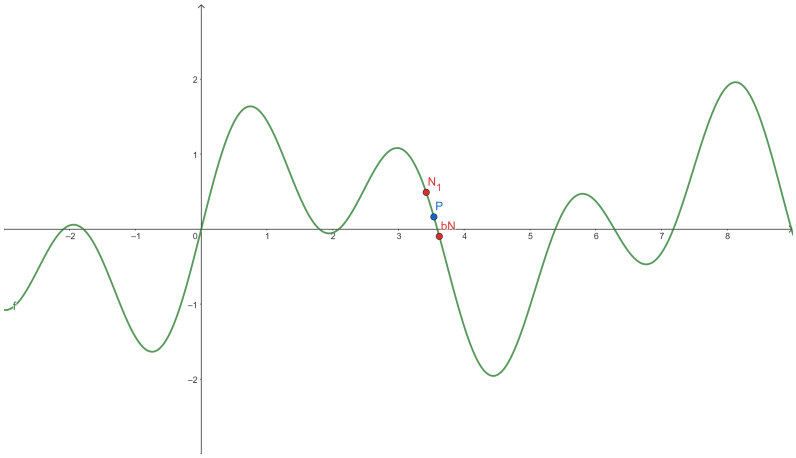



Figure 2.4: Visualization of local search choosing between neighbors of its current potential solution, shown on our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \epsilon [-3, 9]$.

If it is better, it becomes the new potential solution. This process is repeated until the termination criterion is reached or until there are no more better neighbors, as shown in Figure 2.5.

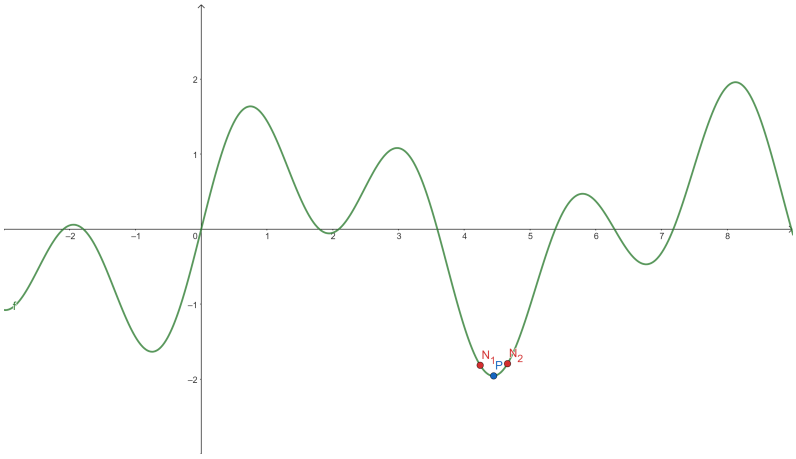



Figure 2.5: Visualization of a potential end state of local search after no more better neighbors can be found, shown on our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \epsilon [-3, 9]$.

### 2.5.2 Strengths and Weaknesses of Basic Local Search

A strength that local search has, its simplicity, is also the origin of its biggest weakness. Local search has a very large chance of landing in local optima if they exist. As a local optimum, by definition, has no better direct neighbors, local search would accept and return it. This is a problem that virtually all divergent local search algorithms try to get rid of or at least make unlikely [GK03, page xi]. The paths taken vary from iteration to stochastics and often produce promising algorithms, also for the problem of computational unfolding, such as simulated annealing.

Even though local search can be viewed as a simple algorithm, it needs certain parameters that are not necessarily simple to define. Especially in the field of computational unfolding, questions can arise. One such question is what is a neighbor. If our search space is, for example, on a number line, we can define a neighbor as a number with a certain distance to our current position. This task gets harder with unfolding as we have no number lines and changes to our state are made by moving mesh faces. Neighbors of a mesh configuration could be defined as all mesh configurations with exactly one mesh face difference or all configurations where the underlying spanning tree $T$ is changed at exactly one edge of the dual graph $D$. Another question that arises is what is defined as a

better neighbor. If we have an objective function, this question is easily answered, but in the case of unfolding, we simply want as few overlaps as possible. So better could be defined as fewer overlaps or it could be defined as a smaller overlap area or fewer branches of the spanning tree $T$ still containing overlaps. These parameters aren't easily defined and there isn't one correct one for all variations of local search. The way these parameters were chosen in this thesis is further discussed and explained in Section 4.

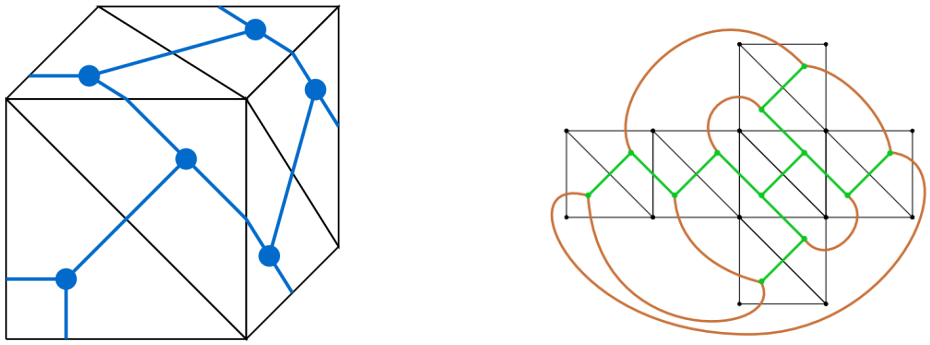
# 3 Related Work

## 3.1 Optimized Unfolding via Simulated Annealing

The most closely related work to this thesis is the paper by Korpitsch et al. [KTGW20]. This section summarizes their work and results and shows similarities as well as differences to this thesis.

### 3.1.1 Paper

Their paper discusses unfolding 3D meshes into a singular coherent patch of planar polygons, similar to this thesis. Analogously, they generate a dual graph $D$ out of their 3D mesh (blue edges and vertices in Figure 3.1(a)), from which they then obtain a minimum spanning tree $T$, denoted as MST, that serves as the basis of the unfolding process. They introduce the concept of Bend-Edges and Cut-Edges. Bend-Edges are unfolded or bent in the unfolding process and are the edges that are displayed as green in Figure 3.1(b). They are defined as dual edge $e \in E_T$, where $E_T$ is the collection of all edges of the MST $T$. Cut-Edges are the edges that are cut apart in the unfolding process and are displayed as brown in Figure 3.1(b). They are defined as dual edge $e \in E_D \backslash E_T$, where $E_D$ is the collection of all edges of the dual graph $D$ and $E_T$ is the collection of all edges of the MST $T$.



(a) (b)

Figure 3.1: (a) Blue dual graph $D$ laid on top of a 3D mesh of a cube. (b) Unfolded 2D patch of a cube mesh. Green and brown represent the edges of the dual graph, while green are Bend-Edges and brown are Cut-Edges. [KTGW20, figure 2]

The way they arrive at their first MST and, therefore, the unfolding, is by giving every edge $e \in E_D$ a random weight between (0,1). This is because for the generation of a reasonable MST weighted graph edges are required and the edges of a 3D Mesh are not naturally weighted. Later, they use the optimization algorithm simulated annealing to adjust these edge weights until an overlap-free unfolding is achieved.

The two main differences between our work are that they add glue tabs to their 2D patches and the optimization algorithm used is simulated annealing. They add glue tabs for real-world papercraft applications, stating: ”Glue tabs are essential as they allow users to build clean 3D models and guide them during the reconstruction process.” [KTGW20, chapter 1]. Glue tabs also increase the difficulty of reaching an overlap-free configuration as the added trapezoidal faces take up additional space.

### 3.1.2 Explanation of Simulated Annealing

The biggest difference is clearly the chosen algorithm. To further clarify their approach, a quick explanation of simulated annealing is needed. Simulated annealing gets its core idea by emulating a process in metallurgy, where a metal is heated to a temperature $T$ and then cooled off according to a cooling schedule $S$ to achieve a structure with a minimum energy configuration [BT93].

The following explanation of simulated annealing is based on [BT93, chapter 1], [KTGW20, chapter 2.2] and [KGV83]. The algorithm keeps a candidate solution at all times, starting off in the case of Korpitsch et al. [KTGW20] with the configuration of the original MST. It also defines a starting temperature $T_{max}$ and a cooling schedule $S$. It iteratively searches for neighboring solutions by changing the weights on dual graph $D$ and by extension the configuration of MST $T$. At each step, the algorithm decides whether the neighboring solution shall replace the candidate solution. If the new solution is better than the old one, it is replaced, and if it is worse, it is replaced with a probability relative to the current temperature $T$ of the system. After every iteration, $T$ is further decreased by a value defined in $S$. The concept of temperature in this algorithm gives a trade-off between exploitation and exploration. Meaning the algorithm explores more of the search space while temperatures are high and exploits better solutions while temperatures are low. The probability for accepting a worse solution is defined as $exp(-\Delta J/T)$, where $\Delta J$ is the cost difference between the two solutions and $T$ is the current temperature of the system. Korpitsch et al. [KTGW20] also introduce a further constant $k_B$ that should be defined and adjusted through experimentation. $k_B$ is not a part of the regular simulated annealing algorithm. Their probability is therefore defined as $exp(-\Delta J/k_B/T)$. Bertsimas and Tsitsiklis [BT93, chapter 2.1] state that the most popular cooling schedule in theory must be of the form $T = \frac{d}{log\, T'}$, where $d$ is a constant and $T'$ is the previous temperature. The algorithm continues until the temperature reaches a predefined value. In the case of Korpitsch et al., the temperature will drop down to 0. The reasons for using simulated annealing over other algorithms are: "Simulated annealing is easy to implement (...) and it is more likely to find an optimal solution compared to a greedy algorithm." [KTGW20, chapter 2.1].

### 3.1.3 Visualization of Simulated Annealing

To Visualize the previous explanation, the base function will once again be used. As stated, simulated annealing keeps a candidate solution or potential solution. This is shown as starting point $P$ in Figure 3.2.

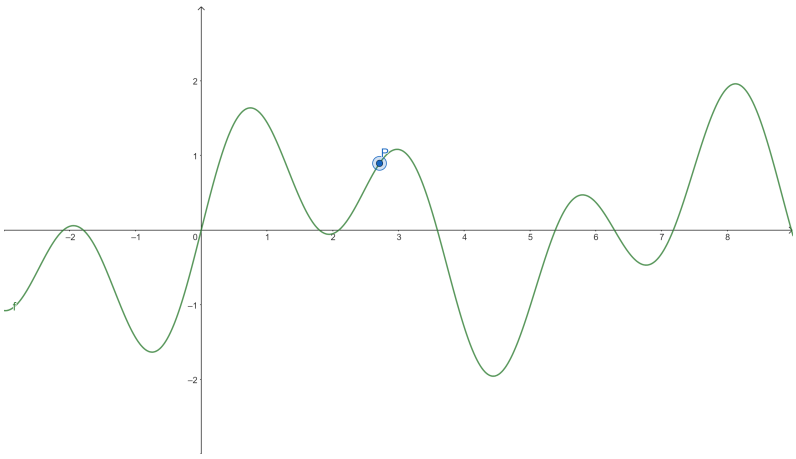



Figure 3.2: Visualization of the starting state of the simulated annealing algorithm over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \in [-3, 9]$.

With a regular local search algorithm, $P$ would now start moving to the left, as that is where the better neighbors are situated. This would continue until it gets stuck in the local minimum at around $x \approx 2$. However, since at the beginning of simulated annealing, the internal temperature $T$ is high, there is a large probability that worse

neighbors will be chosen. This, in turn, can move $P$ to the right and potentially over the local maximum, as shown in Figure 3.3.

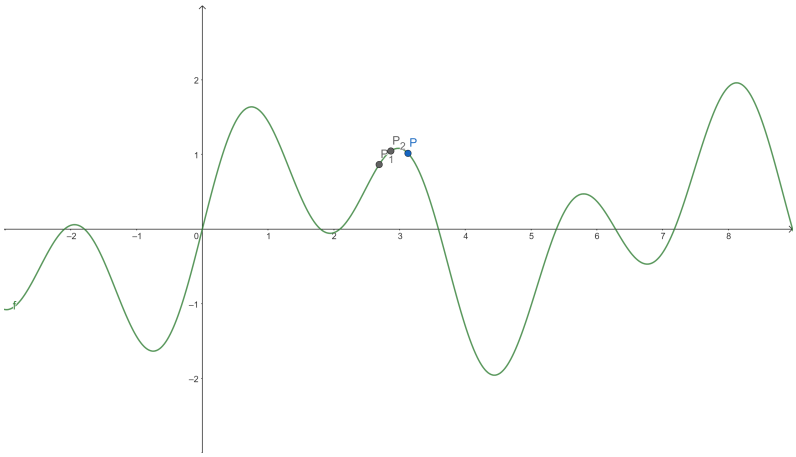



Figure 3.3: Visualization of the possibility simulated annealing escapes local optima by choosing bad neighbors, shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \epsilon [-3, 9]$.

The issue with this algorithm is that it could leave the neighborhood of the global minimum similarly as it left the local minimum. This is why simulated annealing cannot guarantee finding the best solution. When the temperature has sufficiently cooled, the algorithm will end up at a minimum and be unable to leave it again, as shown in Figure 3.4.

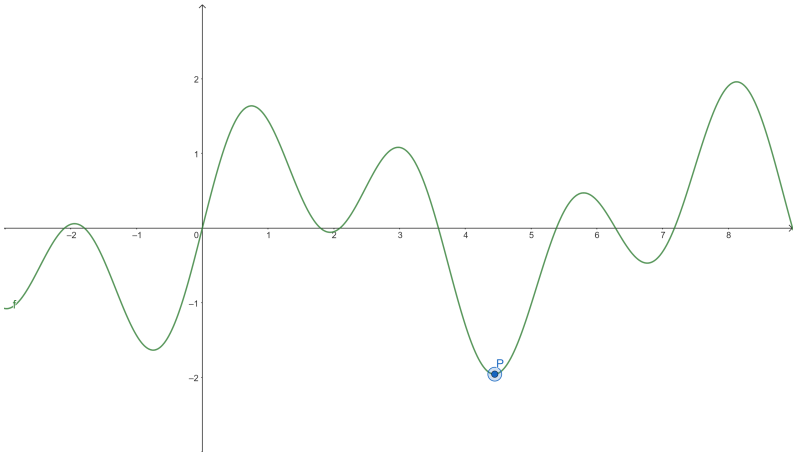



Figure 3.4: Visualization of the end of the simulated annealing algorithm, when the temperature has cooled and minima cannot be escaped any longer, shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \epsilon [-3, 9]$.

The pseudocode in Listing 10.1 is inspired by [BT93, chapter 1] and [KTGW20, chapter 2.2] and demonstrates how the algorithm works.

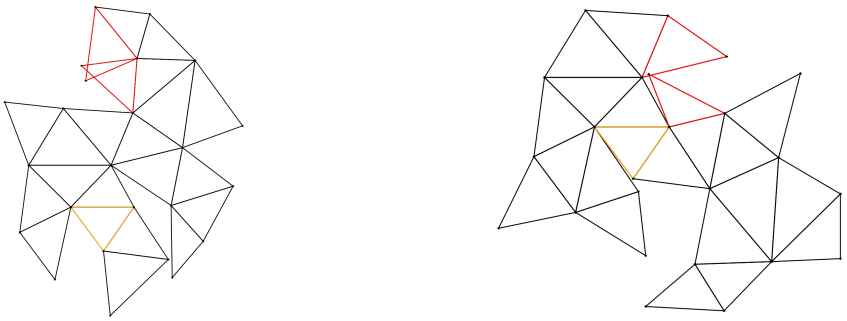
## 3.2 Optimized Unfolding via Genetic Algorithm

### 3.2.1 Paper

A second work closely related to this thesis is the paper by Takahashi et al. [TWS+11]. Their approach to achieving overlap-free unfoldings differs a little more from the approach of this thesis. They unfold their meshes into multiple incoherent patches, which are then, in a second step, stitched back together to reach a coherent 2D

configuration using a genetic algorithm.

One thing they do is differentiate between local and global overlaps [TWS+11, chapter 3.3]. They define a local overlap as faces overlapping that are directly connected via a vertex. As you can see in the example Figure 3.5(a), in a local overlap scenario, the overlapping faces share a vertex. While on the right, in Figure 3.5(b), the overlapping faces share no vertex and are therefore overlapping globally.



(a) (b)

Figure 3.5: Overlapping polygons are marked red. (a) An example of a local overlap on an unfolding. (b) An example of a global overlap on an unfolding.

They state that this distinction makes sense as local overlaps can be found and avoided before the unfolding process takes place. Any vertex where the "sum of the corner angles around that vertex exceeds $2\pi$" [TWS+11, chapter 3.3], $2\pi$ being 360 degrees, must have at least two cut edges. This becomes clearer when looking at Figure 3.5 again. As, in the left image, the only cut originating from the shared vertex of the red faces, was made between the two red faces. This necessarily results in an overlap as the vertex only has 360 degrees of overlap-free space around it and the connected faces exceed that angle. On the other hand, global overlaps cannot be easily detected pre-unfolding [TWS+11, chapter 3.3].

Just as is done in this thesis and in the paper by Korpitsch et al. [KTGW20], Takahashi et al. [TWS+11, chapter 4.1] also create a dual graph $D$ out of their 3D mesh. They then generate a spanning tree $T$ which is used for the unfolding process. Their way of getting from $D$ to $T$ differs slightly from Korpitsch et al. [KTGW20], but is not further relevant for the understanding of their procedure apart from the fact that they create multiple spanning subtrees $T_N$ instead of one spanning tree $T$. This leads to them having many smaller disconnected patches of faces, rather than a singular large coherent patch. They then remove global overlaps and any local overlaps, that remained undetected pre-unfolding, by "tracking the path between the [overlapping] faces on the unfolded patch, and (...) cutting the patch along some edge on that patch" [TWS+11, chapter 4.2]. This translates to them simply cutting their patches apart further when overlaps are encountered, creating more and smaller, overlap-free patches in the process.

The next step is stitching the many small patches together into as few patches as possible [TWS+11, chapter 5]. For that, an ID is given to all the currently disconnected triangle edges as shown in Figure 3.6. Triangle edges that share an edge on $D$ receive the identical ID. If the theoretical ID pair is already on the same patch the ID is omitted, as one cannot stitch together a patch with itself. Now you know which face can be connected to which other faces via which edge. As the original 3D mesh may not be altered, every face edge has exactly one other face edge it can potentially connect with. During this process, any potential local overlaps can once again be identified and avoided with the technique of looking at vertex corner angles.

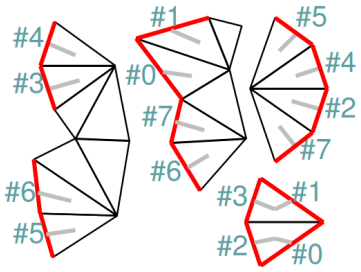



Figure 3.6: Multiple small polygon patches with IDs on the stitchable edges. Edges with identical ID can be stitched together. [TWS+11, figure 7]

### 3.2.2 Explanation of Genetic Algorithm

The order in which the patches are stitched together is determined and optimized by a genetic algorithm [TWS+11, chapter 5.2]. A genetic algorithm is a type of optimization algorithm that stems from the ideas of natural selection and evolution [HMR16, chapter 1]. It usually relies on a loop where first the fitness of all *living* individuals (equivalent to potential solutions in regular local search algorithms) are assessed with a fitness function (equivalent to objective or cost functions in regular local search algorithms), to check if the termination criteria have been met. If the termination criteria are not met, individuals are modified through crossover and mutation techniques, simulating genetic mutations in natural evolution between generations. A selection technique is then used to choose which individual or individuals will be used for reproduction. Reproduction in this case means, which individuals get to produce how many of the next iteration of individuals. Here usually a selection technique is used, that will give fitter individuals a higher chance of reproduction, with the plan of eventually creating a population of high-quality individuals that reach the optimum of the fitness function [HMR16, chapter 2]. In a genetic algorithm, the search space is defined by chromosomes. Each chromosome is a string which represents its individual. For example Takahashi et al. [TWS+11, chapter 5.2] define a chromosome as a sequence of edge IDs in the order they should be stitched together.

Takahashi et al. [TWS+11, chapter 5.2] begin by randomly generating the starting chromosome, which for the example Figure 3.6 would be 8 characters long, as there are 8 different possible stitches. Their fitness function $f$ is defined as follows [TWS+11, chapter 5.3]:

$$f = \lambda_p N_p + \lambda_l R_l + \lambda_m R_m + \lambda_b R_b \tag{3.1}$$

The different $\lambda$ are weights for the four other variables. $N_p$ is the number of disconnected patches, $R_l$ is the number of faces not connected to the biggest patch divided by the total number of faces, $R_m$ is a number between [0,1] depicting the relative margin around the unfolding and $R_b$ is "$E_b/E_t \epsilon$ [0,1] where $E_b$ is the average number of edges between each pair of the duplicated boundary edges and $E_t$ is the total number of boundary edges, respectively" [TWS+11, chapter 5.3]. The two first variables attempt to minimize the number of patches, while the second two attempt to make better use of the space on a sheet of printing paper. Since the primary goal is managing to reach one coherent patch, $\lambda_b$ was weighted 100 times and $\lambda_m$ 10 times less than the other two. The Equation 3.1 shows another big difference between the approach of Takahashi et al. [TWS+11] and this thesis, as in this thesis the optimization of the used paper space is disregarded.

The next step is generating offspring. There are two main ways to alter the parents to create offspring: mutation and crossover. Crossover involves taking two parent chromosomes and crossing them. This is achieved by copying parts of the parents' strings and adding them together to form a new child chromosome[HMR16, chapter 3.B]. Mutation, on the other hand, needs only a single parent chromosome and changes small parts down to single characters in a chromosome to create the child chromosome [HMR16, chapter 3.C].

After the genetic algorithm does its work, Takahashi et al. [TWS+11, chapter 6] typically ended up with a singular unfolded patch. If they encountered certain global overlaps that could not be resolved, they would still stitch them together and then cut them apart at a different location to attempt a resolution. This process would be repeated until a single coherent patch could be achieved.

# 4 Theoretical Solution

In this section the algorithms that were chosen to solve the before defined problem are described and explained.

## 4.1 Iterated Local Search

The first algorithm that will be used for unfolding 3D meshes is called iterated local search [RLMS00, page 2]. As the name suggests, it is a variant of the local search algorithm. Iterated local search combines the regular local search algorithm with random perturbations to prevent getting stuck in local optima. This makes it particularly useful for tackling complex problems.

At its core, iterated local search is an algorithm that iterates the local search algorithm[RLMS00, page 2]. This means that a normal local search is executed to completion as shown in Figure 4.1.

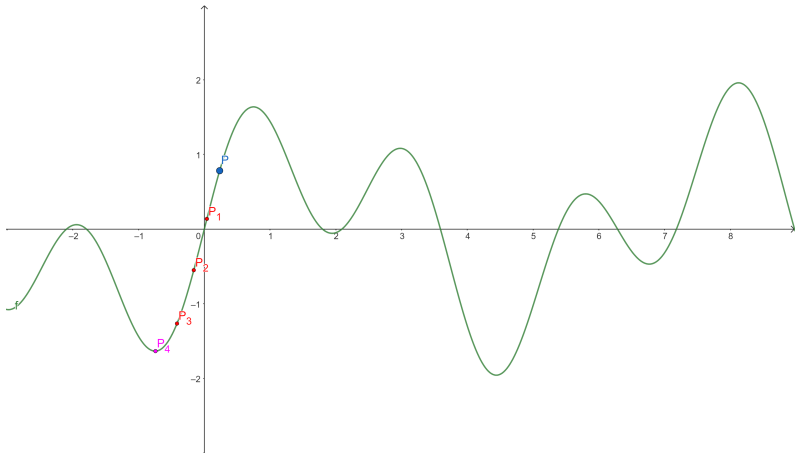



Figure 4.1: Visualization of the first iteration of iterated local search. Starting from $P$ until reaching $P_4$ where the first iteration terminates. Shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \epsilon [-3, 9]$.

Then, a perturbation is made: Changing either the current solution or restarting from a new starting point. These changes or restarts can be of a random nature or can be guided by preselected factors to enhance the chances of finding an optimal solution. This concept is shown in Figure 4.2.

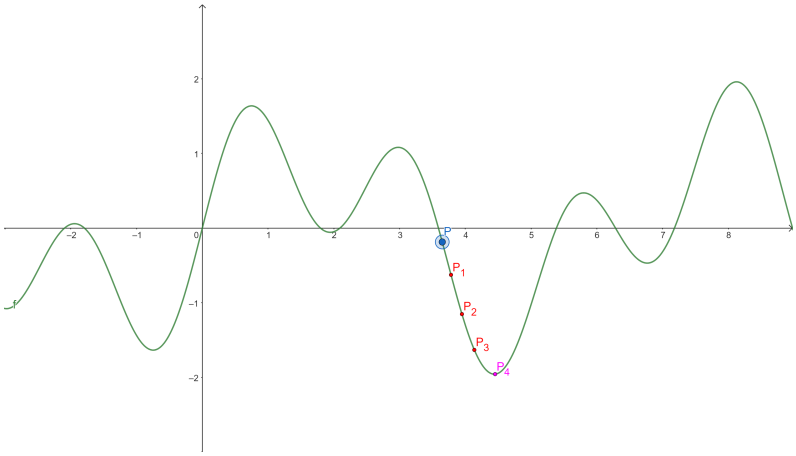



Figure 4.2: Visualization of a following iteration of iterated local search. Starting from $P$ until reaching $P_4$ where global optimum is found. Shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x\ \epsilon\ [-3, 9]$.

"The simplest possibility to improve upon a cost found by LocalSearch is to repeat the search from another [random] starting point."[RLMS00, chapter 2.2]. This means that we randomly start our local search algorithm from new starting positions in the search space with the hope of eventually reaching a global optimum. The good thing about this version of the algorithm is that it guarantees finding a global optimum if one exists, as one will eventually search the entire search space. On the other hand, it can be incredibly slow, as in the worst-case scenario, nearly all configurations of the 2D mesh must be tried as starting positions, making this algorithm no better than brute force.

Another idea is to use some sort of heuristic to better define where to restart from. This can help to find some sort of middle ground, where the strength and weakness of the algorithm become more balanced. The tricky part, of course, is what kind of heuristic to use that can lead the algorithm in a fruitful direction upon restart. "By how much should the perturbation change the current solution? If the perturbation is too strong, ILS may behave like a random restart, (...) if the perturbation is too small, the local search will often fall back into the local optimum just visited"[RLMS00, page 12].

Some of the possibilities pondered during the writing and implementation of this thesis include:

- Randomly moving a singular randomly chosen face.
- Starting from an overlapping face, move up the spanning tree $T$ and move the highest movable face.
- Starting from each overlapping leaf of $T$, move up the tree and move a randomly chosen face.

These heuristics are all options that are to be looked at to find a suitable one. In theory, the first one is very easy and fast to implement but changes the overall structure of $T$ and with that the 2D mesh only very minimally. This could mean that it more easily enables the algorithm to get stuck over and over again in the same local optimum. The second one alters the spanning tree a little more, increasing the chances of escaping a local optimum. Option three is the most costly and complex to execute but can heavily change the structure of $T$. The problem with it is that it has the potential of shuffling our mesh so extensively that it is very similar to just doing a random restart.

Pseudocode showing how iterated local search works is shown in listing 10.3.

## 4.2 Stochastic Hill Climbing

The second algorithm attempting the same goal as iterated local search is stochastic hill climbing. Regular hill climbing can simply be viewed as another name for the basic local search algorithm, where the algorithm always

climbs the steepest hill or, in other words, makes the best next move as described in chapter 2.5. Stochastic hill climbing introduces randomness into the algorithm to try and avoid getting locked into local optima. This is done in a similar way to simulated annealing from Section 3.1. The algorithm keeps a candidate solution like its non-stochastic sister algorithm. Contrary to regular hill climbing or local search, it doesn't necessarily always choose the best neighbor as the next candidate solution [Rus21, page 131]. It calculates a probability $N_p \epsilon$ [0,1] for each neighbor, which dictates how likely it is to choose each neighbor as its next solution. The better a neighbor, the higher $N_p$. Normally, the algorithm does not allow worse solutions to be chosen and assigns them a probability of 0: $N_p = 0$. However, in this thesis, the algorithm is changed to give worse solutions a probability $N_p > 0$. The reason for this is that it further increases the chances of the algorithm to escape local optima while sacrificing a little bit of performance. For a visual representation of the algorithm Section 3.1.3 shows off the functionality in Figures 3.2, 3.3 and 3.4 as it shares many similarities with simulated annealing.

The algorithm can be seen as pseudocode in listing 10.4.

## 4.3 Adaptive Step Size Random Search

Adaptive step size random search, from here on abbreviated as *ASSRS*, is the least straightforward algorithm of the bunch. The version of this algorithm used in this thesis varies a bit from the regular version. Normally, the algorithm works in a way that it searches the search space at a given distance around the current candidate solution. All solutions at this predefined distance can be looked at as neighbors. ASSRS chooses the best neighbor as the new candidate solution and so on. The special part is that it always checks for neighbors at two different distances. The first distance being the current *step size* $S$ and the other being slightly larger. If the slightly more distant neighbors prove to be better, this step size gets set as the new step size. If after a few iterations the larger step size is always worse, the opposite happens and it gets iteratively reduced [SS68, page 272].

In this thesis, some basic concepts of ASSRS are changed. Firstly, step size $S$ will be defined as the gravity of changes made in an iteration. Meaning, if we are at the lowest step size, we will move the lowest, or closest, polygonal face of our mesh per iteration, this is explained in more detail in Section 5.3. The second difference will be that the step size starts large and gets progressively smaller. This concept is visualized in Figures 4.3, 4.4 and 4.5. The idea behind this is that the algorithm will first make large moves that drastically change the layout of the mesh and then get continuously more exact and subtle as we close in on an optimal configuration. This tactic is inspired by the simulated annealing algorithm described in Section 3.1.2.

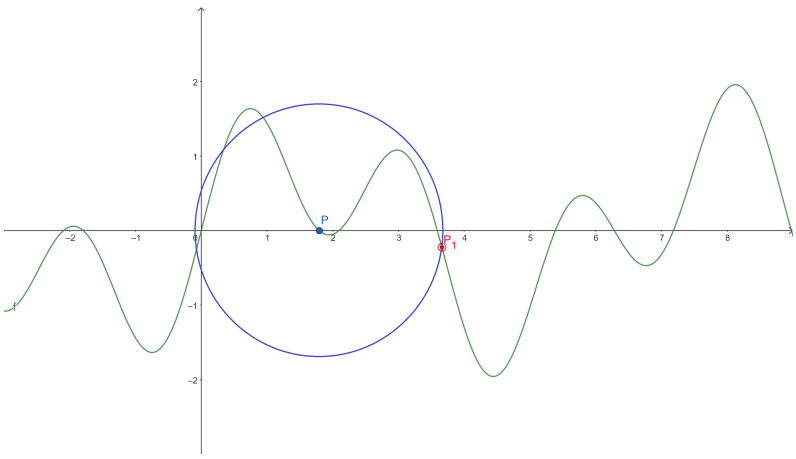



Figure 4.3: Visualization of the first step of ASSRS, where the step size is still very large as indicated by the blue circle. Shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x \epsilon [-3, 9]$.

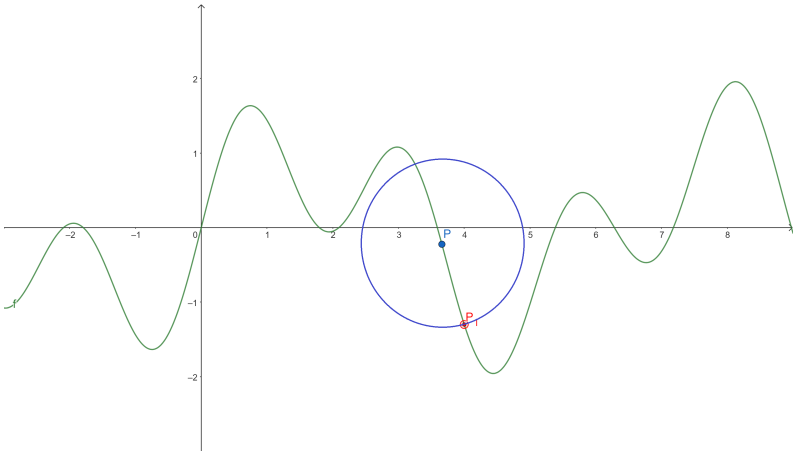



Figure 4.4: Visualization of an intermediate step of ASSRS. Shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x\ \epsilon\ [-3, 9]$.

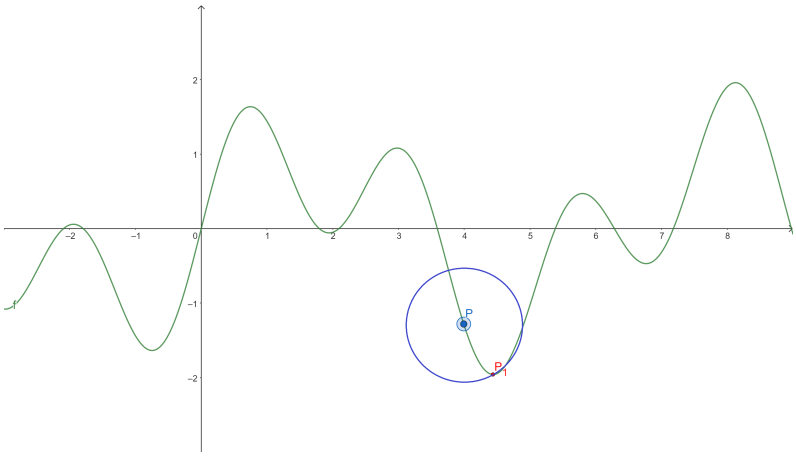



Figure 4.5: Visualization of the last step of ASSRS, where the step size is very small as indicated by the blue circle. Shown over our base function: $f(x) = sin(x) + sin(\frac{5}{2}x),\ x\ \epsilon\ [-3, 9]$.

ASSRS is definitely a bit more complex than the others, but it promises the potential for quickly solving the problem at hand. The biggest issue with it is getting stuck in local optima, as it is basically a hill climbing algorithm with varying definitions of neighborhood. Therefore, a good strategy for escaping potential local optima is paramount for this algorithm.

# 5 Implementation

This chapter discusses the practical implementation of the algorithms. It shows what changes were made to the stock algorithms to better accommodate the problem at hand and explains why and how these changes are beneficial to this thesis. This chapter does not yet delve into the results obtained from running the algorithms but rather focuses on the ideas and expectations ahead of testing.

First, it is also necessary to explain the basic implementation ideas that are the common to all of the algorithms. Each algorithm has been constructed in similar fashion. To begin, the algorithms receive a randomly unfolded version of a prior 3D mesh, which is obtained according to Section 2.2. This implies that it contains a plethora of overlaps. Additionally, each algorithm has knowledge of the dual graph $D$, as seen in Section 2.2, to correctly move faces.

All iterations of the algorithms follow a similar structure. The intricacies of the individual tasks will be described in detail in the following sections. however, a general overview of the shared processes still makes sense. The first process, referred to as $determineLocalMinima$ from now on, is always to check if an algorithm has become stuck in a local optimum. If a local optimum is determined, the algorithm will complete a predefined action to exit that local optimum, such as randomizing the unfolding. For the iterated local search algorithm, this is the only way to avoid local optima, while for the others, this serves as more of a fail-safe, as they should never become permanently stuck. The second process is selecting a face on the unfolding to work with further. This is done equivalently for all algorithms. They always choose a random face that is involved in an unresolved overlap and is movable. *Movable* entails not currently being connected to all your neighboring faces from the original 3D mesh and also not disconnecting the coherent 2D patch if a move were to be made. The third process, referred to as $takeStep$ from now on, is where a move is made. This process differs heavily between the algorithms. In general, the idea here is evaluating possible moves, starting from the currently selected face, and deciding whether or not to make them according to the algorithm's logic. If no move is made on the selected face, most algorithms then climb up the spanning tree and retry moving until they reach the root node, at which point the process is abandoned and a new iteration is started. If these iterations at any point reach a global optimum, defined as zero overlapping faces, the algorithms terminate.

## 5.1 Iterated Local Search

The iterated local search algorithm is the simplest one to implement. For the purposes of this thesis, two versions of the algorithm were implemented, which can be switched between by changing the boolean variable: $m\_randomPerturbations$. As the variable name suggests this changes the way the algorithm perturbs its state upon reaching a local optimum.

Both versions share an equivalent $takeStep$ process, in which the currently selected face is attempted to be moved. If the best of these possible moves promises fewer overlaps than the current state, the move is made. Otherwise, the spanning tree is climbed and at each new face the same comparison is made, until a move can be completed. If no move was better than the current state, when the root node is reached, the algorithm iterates and selects a new face to continue from. This part of the algorithm, before perturbation, is an implementation of the basic local search algorithm, from Section 2.5.1. For each fruitless iteration the variable, $m\_stepsSinceMovement$, is increased by one. If $m\_stepsSinceMovement$ reaches a threshold, which is defined as $\frac{1}{10}$ times the number of faces on the mesh, the algorithm assumes it is stuck in a local optimum.

The $determineLocalMinima$ process, triggered by $m\_stepsSinceMovement$ reaching the threshold, is where the two versions of the algorithm diverge. The randomly perturbing algorithm simply re-unfolds the 3D mesh and deletes the old 2D configuration that is currently stuck in a local optimum. This is the equivalent to a completely random restart, as discussed in Section 4.1. The other version of the algorithm uses knowledge of its state to complete perturbations. Two different options were explored here. The first, and ultimately discarded, was attempting to move a parent face of a currently overlapping face. The idea was that one would climb up the tree from an overlapping face until the highest still movable face had been identified. Then, that face would be moved randomly, creating a similar, yet hopefully different enough, perturbation of the previous state. However, this approach presented an issue: If the final remaining unsolvable overlap is close to the root node, there might not be a movable parent face. This leads the algorithm to get eternally stuck in its local optimum, making a sizable portion of problems unsolvable. The second option that was tried is taking a random face, whether overlapping or not, and moving it. This can create perturbations that are minuscule or quite large, but always stay close to the state before. This approach alleviates the problem of getting stuck in a local optimum forever, but relies more heavily on randomness to achieve a sensible perturbation. Even though the second version of the algorithm tries to be less reliant on randomness, a fail-safe was still implemented, that will make a random restart happen, if no solution was found after 10 perturbations.

## 5.2 Stochastic Hill Climbing

During the writing of this thesis, while testing the algorithms, the idea of adding an adaptive aspect to the stochastic hill climbing algorithm arose. Consequently, this section is split up into two mostly equivalent algorithms. They diverge in the fact that Section 5.2.2 describes a version of stochastic hill climbing, where the probability values change over time based on the number of remaining overlaps.

### 5.2.1 Regular Stochastic Hill Climbing

The stochastic hill climbing algorithm has a $determineLocalMinima$ process that randomly resets the entire unfolding. Since stochastic hill climbing has ways to leave local optima of its own power this process should never trigger. As a fail-safe, it is defined in a way that the random restarts happen, after the algorithm could not find a global optimum after a massive number of steps taken.

The key process for this algorithm is the $takeStep$ process. Here, the algorithm checks for possible neighbors at its current node. Neighbors, being faces that the current node can be moved to without breaking apart the singular coherent patch of polygons. Contrary to iterated local search however, stochastic hill climbing looks at all neighbors as potential moves, even worse ones. It assigns every possible move that decreases the total amount of overlaps a number equal to that decrease. It assigns every move that increases the amount of overlaps the number 0.5 and every move, where the amount of overlaps remains the same, the number 1. This can be seen as the algorithm valuing bad moves as an improvement of half an overlap and indifferent moves as an improvement of one overlap. The algorithm then chooses a neighbor with a probability that is proportional to the number assigned. A further condition must be added, so that the algorithm does not select bad moves too often. Given that faces on our 2D patch have at most two possible neighbors, it can occur that both are worse or equivalent to the current configuration quite often. In that case, a 100% chance of the algorithm picking one of them is undesirable. Therefore, if no better neighbor is found for the current node, the number 10 is assigned to making no move at all, significantly reducing the number of bad and indifferent steps taken. When no step is taken or no neighbor is found for a node, the spanning tree is climbed until a move is made or until the root node is reached. Upon reaching either of these two criteria the algorithm iterates.

### 5.2.2 Adaptive Stochastic Hill Climbing

What can be perceived when running the regular stochastic hill climbing algorithm, is that when overlap numbers get small and therefore, often no better neighbors are found due to being stuck in a local optimum, the algorithm

tends to lose time by making no moves or indifferent moves, instead of making bad moves to escape. For this reason, the probability that is assigned to making no moves at all, is reduced, as overlap numbers reduce, in the adaptive stochastic hill climbing algorithm. Initially, the algorithm maintains 10 as the value assigned to making no moves, as is the case in the regular stochastic hill climbing algorithm. When the number of overlaps reaches less than half the amount of mesh faces this value is changed to 5 and when the overlaps are fewer than five the value is changed to 3.

## 5.3 Adaptive Step Size Random Search

The fundamental concept behind the ASSRS algorithm is, as described in Section 4.3, to change step size while iterating. As previously noted, in this thesis the change in step size has been inverted. This means that instead of increasing step size, as is the case in regular ASSRS, the step size is decreased. Further, the algorithm has been simplified. In the implementation used for this thesis, the algorithm always selects the move that decreases the amount of overlaps and is located furthest up the spanning tree. Consequently, the algorithm attempts to solve the overlap issue top down, effectively decreasing step size.

The implementation of ASSRS shares its $determineLocalMinima$ process with the two stochastic hill climbing algorithms. In theory, this process should not be necessary, as it is implemented as a fail-safe. Equivalent to the other algorithms, it randomly resets the unfolding after a large amount of iterations, essentially providing a fresh start.

The $takeStep$ process is where the main work of the algorithm takes place. In the same manner as the others, it randomly selects an overlapping face and then checks if any of its neighbors will decrease overall overlap numbers. If any of them do, it selects the best one, remembers it and climbs up the spanning tree towards the root node. It repeats the neighbor checking step at every node and replaces the remembered neighbor whenever there is a new good one. It does this until it reaches the root node and then executes the last move it remembered, effectively making the highest possible good move. This works well until the algorithm gets stuck in a local optimum. Then an escape is implemented. The algorithm also always remembers the highest possible move, regardless of whether it was good or bad. If it has not improved the overlap numbers for ten iterations it executes this move. This can lead the algorithm to potentially escaping the local optimum and finding a solution.

# 6 Results

This section serves as an overview and visualization of the collected data, after repeatedly unfolding a multitude of 3D meshes, using the algorithms from Section 5. The data was collected by unfolding ten distinct meshes, each with four different resolutions, and recording the time taken by the algorithms. The meshes had resolutions of 100, 200, 400 and 600 faces. The highest resolution version of each mesh is provided in Section 10.2. A failed attempt was defined as an algorithm taking longer than $\frac{faces^2}{500}$ seconds, $faces$ being the number of faces the mesh has. This leaves algorithms with a maximum time of 20 seconds for 100 faces, 80 seconds for 200 faces, 320 seconds for 400 faces and 720 seconds for 600 faces. If an algorithm eclipses these times, the attempt was abandoned and noted as a failure. Failures were not considered in calculating average solve times. Each algorithm completed this task 20 times. This sums up, $20\ iterations * 10\ meshes * 4\ resolutions$, to 800 unfoldings per algorithm. What must also be noted, is that the non-random iterated local search algorithm version is not featured in the results, as it performed worse than its random counterpart in every preliminary test.

For the purposes of this section, adaptive stochastic hill climbing will be referenced as ASHC, stochastic hill climbing will be referenced as SHC, iterated local search as ILS and adaptive step size random search as ASSRS.

All four algorithms performed acceptably, time-wise, over 100-faced meshes, when looking at Figure 6.1. ASHC with an average solving time of 2.483 seconds was the fastest. SHC, with an average time of 2.727 seconds, was slower by 9.8%. ILS had an average time of 6.846 seconds and was 175.7% slower. ASSRS took an average time of 7.13 seconds and was 187.0% slower. A more problematic viewpoint is acquired, when looking at the algorithm's respective fail rates, as portrayed in Figure 6.2. For 100 faces, a fail was, as introduced in Section 6, defined as an unfolding eclipsing 20 seconds. ASHC and SHC, with respective fail rates of 1.0% and 4.0%, were able to solve the majority of tasks. ILS could not unfold 23.5% of meshes in under 20 seconds, while ASSRS could not manage 57.0%.

ASHC remained the fastest algorithm for 200-faced meshes, completing an average unfolding in 11.141 seconds. SHC was slower by 27.4%, with 14.196 seconds. ILS and ASSRS were now clearly starting to heavily drop off, as their gap increased to, 328.1% with 47.707 seconds and 272.8% with 41.524 seconds, respectively. It looks even worse for the latter two algorithms, when looking at the fail rates. ASHC and SHC, with respective fail rates of 3.0% and 4.5%, were able to solve the majority of tasks once again. ILS failed 78.0% of its attempts and ASSRS 87.0%. A fail for 200 faces was defined at over 80 seconds.

Over 400-faced meshes, ASHC was fastest again. It took an average of 64.939 seconds. SHC was third fastest, with an average time of 77.656 seconds, 19.6% slower than ASHC. ILS performed worst. With an average time of 130.623 seconds, it was 101.1% slower. ASSRS took an average of 71.079 seconds, making it 9.5% slower. Even though it looks as if the two hill climbing algorithms have been caught up to, Figure 6.2 tells a different story. 320 seconds was the cut-off point for a successful unfolding. ASHC failed only 1.5% of its attempts. SHC failed 5.0%. With 99.0% and 99.5%, ILS and ASSRS failed almost every attempt. ILS, over the 200 unfoldings, had two successes, ASSRS had one.

With 600-faced meshes, ILS and ASSRS did not record a single successful unfolding. Among the remaining algorithms, ASHC was faster by 13.7% with 165.755 seconds, compared to 188.424 seconds for SHC. Unfoldings, longer than 720 seconds were failures. ILS and ASSRS had fail rates of 100%. ASHC and SHC each recorded a fail rate of 6.5%.

ASHC and SHC clearly emerged as the best performing algorithms. Over all unfoldings, as seen in Figure 6.3, ASHC had an average time of 61.079 seconds, while SHC had 70.751 seconds. This makes ASHC 15.9% faster than SHC, on average.

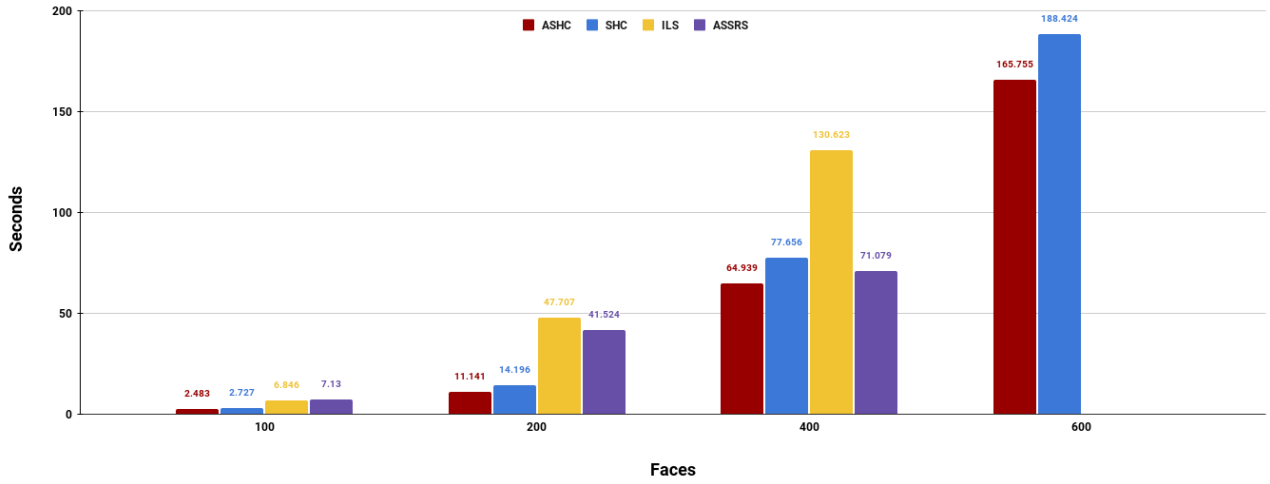



Figure 6.1: Chart showing the average unfolding times for the four algorithms, dependent on the number of faces of the original mesh.

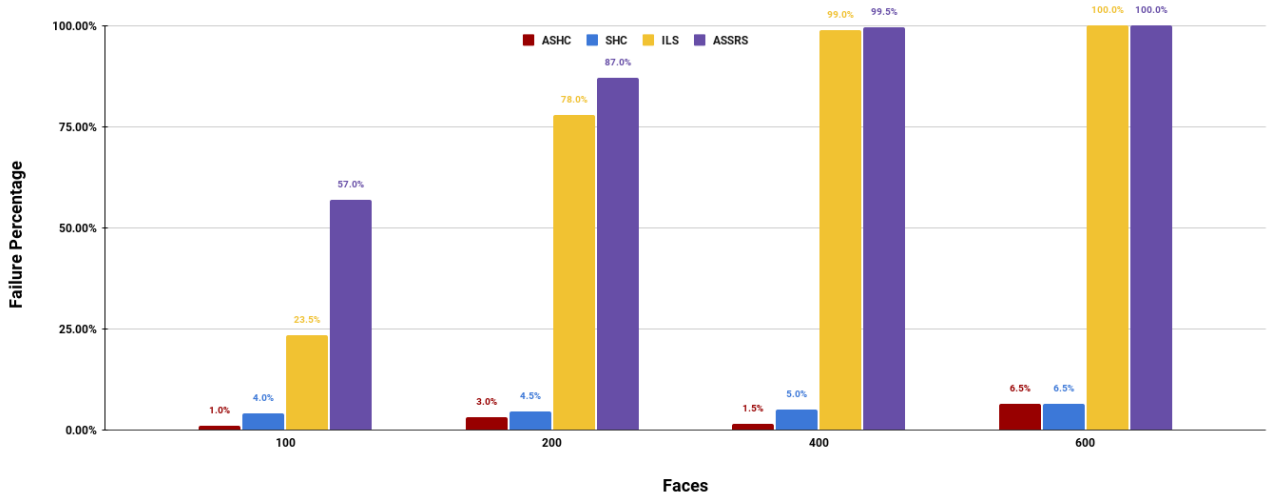



Figure 6.2: Chart showing the failure rates in percent for the four algorithms, dependent on the number of faces of the original mesh.

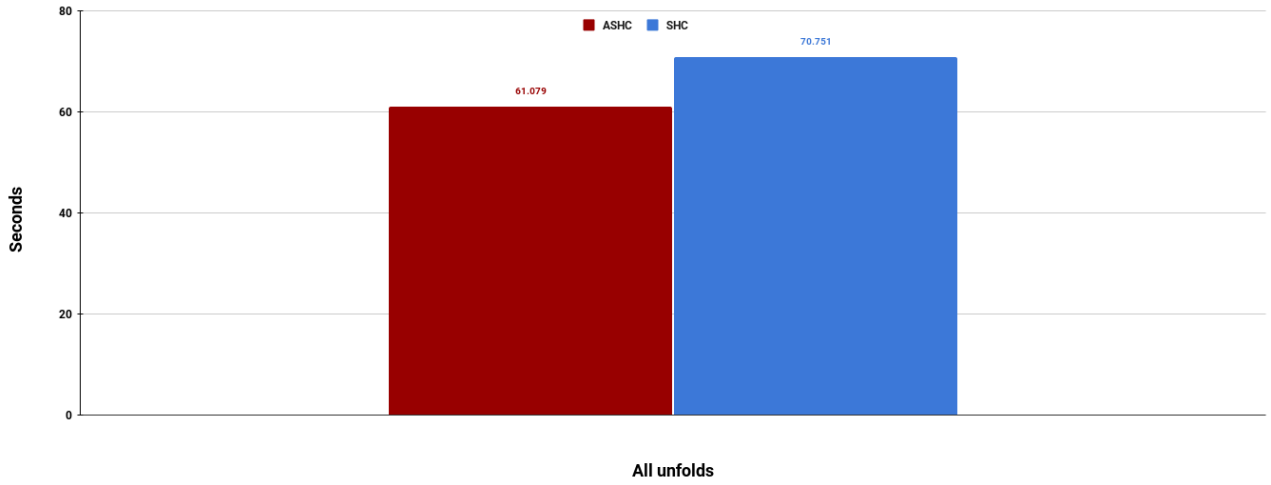



Figure 6.3: Chart showing the average time taken for an unfolding over all 800 unfoldings, for ASHC and SHC.

# 7 Analysis

In this section, the algorithms presented in Section 5 are analyzed based on the gathered results from Section 6. The algorithms will be discussed in order of increasing performance.

The worst performing algorithm by a significant margin was adaptive step size random search, ASSRS. With a fail rate of 87% for 200 faces, 99.5% for 400 faces and 100% for 600 faces, the attempts on 100 faces are the only ones that can really be analyzed. The issue with such high fail rates is that the algorithm's average runtime would be much higher if there was no cutoff point. Even within the 43% successful attempts on 100-faced meshes the algorithm still recorded the highest average runtime with 7.13 seconds. The issue this algorithm has, is even though it unfolds the spanning tree top down, it is still a relatively basic local search algorithm. Meaning, it always selects the best neighbor to move on. The definition of the best neighbor is simply changed to the neighbor that reduces overlaps, that is also the highest up the spanning tree. Algorithms that rely solely on local search, without ways to escape local optima, are highly inefficient for problems such as computational unfolding, as there are huge amounts of local optima. To avoid this, the clause, as described in Section 5.3, was added that forces the algorithm to make the highest possible move, if it's good or bad, after not improving for 10 iterations. While the concept is sound, its execution falls short. The issue is, that the algorithm can get stuck in a loop of moving the top movable face, hardly changing the unfolding, then not being able to make good moves and then moving the top movable face back. This makes ASSRS the only algorithm of the four to regularly become stuck with no way to get out. The problem could be solved or at least lessened, by changing this clause. If the algorithm would just make the worst possible move that is up the spanning tree, instead of the highest possible move, it would have much better chances of escaping local optima.

The second algorithm that did not perform well was iterated local search, ILS. With a significantly lower fail rate and slightly lower runtime on 100 faces it is clearly superior to ASSRS. ILS, as seen in Figure 6.1 has higher unfold times than ASSRS on 200 and 400 faces, but those results can be disregarded, as they entailed such high fail rates that they have no statistical significance. Stock ILS, with random restarts, was anticipated to perform inadequately. It is a heavily luck-based algorithm that relies on a starting point, which allows it to avoid becoming stuck in a local optimum. The potential of ILS came from the non-random version, which makes educated perturbations. The problem was that no such perturbation schedule was found in this thesis. The ones that were tested, always performed worse than regular ILS. It must be stated that even with a good perturbation schedule, it is still expected to perform worse than SHC and ASHC. The clear positive of ILS is its simplicity. It is as close to a basic local search algorithm as one can get while guaranteeing escape from local optima. The other good quality is that ILS will always find a solution if one exists. The problem being, that the time spent could be just as high as brute forcing the problem, which explains the high fail rates.

SHC is a very solid algorithm for computational unfolding. It outperforms the other two previously named algorithms by a massive margin. It has a success rate that is consistently at 93.5%-96%. Even though the way it escapes local optima is very simple, it proved to be quite effective. Unfortunately, SHC cannot be reasonably compared to ILS or ASSRS, apart from stating that it is superior in every way, as the two algorithms performed so poorly.

ASHC was clearly the best-performing algorithm. It is nearly identical to SHC but adapts the probability of making bad moves relative to the amount of overlaps remaining. The idea behind it was to minimize time loss, when regular SHC attempts to make good moves, even though there are none or very few left. An overall increase in performance of 15.9% was achieved, compared to SHC. With a success rate between 93.5% and 99%,

it failed the least and was also consistently the fastest by relatively large margins, between 9.5% and 328.1%. The amplitude of adaptive probability decreases and the overlap thresholds for them, were chosen with minimal testing. This means that with a high probability there is still quite a bit of performance to be extracted from this algorithm, if one were to optimize the adaptiveness of ASHC.

Comparing ASHC to the algorithms used by Korpitsch et al. [KTGW20] and Takahashi et al. [TWS+11] proves rather difficult, as the conditions are not equivalent, as discussed in detail in Section 3. However, a rough comparison can be made. Simulated annealing, when averaging the unfold times of all meshes that had somewhere between 300 and 400 faces, took an average unfold time of 1245.8 seconds [KTGW20, table 1]. This compared to the average unfold time ASHC had for 400 faces, which is 64.939 seconds, is a significant difference. It must be stated that Korpitsch et al. did have a more complex problem to solve, as they added glue tabs to all their unfoldings. Still, the difference in runtime is large and shows that ASHC seems better suited for computational unfolding. Comparing ASHC to the genetic algorithm employed by Takahashi et al. is even more challenging, as they chose a very different approach to unfolding, as described in Section 3.2.1. Still, when averaging the runtimes of all meshes between 300 and 400 faces that found a coherent patch of polygons, one receives 34.2 seconds [TWS+11, table 1]. The average face count of that calculation is 335. ASHC managed 200 faces in an average 11.141 seconds and 400 faces in 64.939 seconds. This puts the genetic algorithm and ASHC on relatively similar runtimes. One would have to run ASHC on their exact meshes to get a concise answer to which algorithm is more efficient.

# 8 Conclusion and Future Work

In conclusion, this thesis attempted to find, implement and analyze optimization algorithms to effectively and efficiently unfold 3D meshes, or more accurately remove overlaps from initial unfoldings. The algorithms that were chosen were: iterated local search (ILS), stochastic hill climbing (SHC), adaptive step size random search (ASSRS) and adaptive stochastic hill climbing (ASHC).

ILS combines the regular local search algorithm with random perturbations to avoid getting stuck in local optima. Despite the theoretical guarantee of reaching a global optimum, ILS was inefficient and largely ineffective. It had a large and rapidly increasing fail rate, and showed significantly longer runtimes than SHC or ASHC. In future work, more educated perturbation strategies could be explored to attempt to find competitive versions of ILS. Ultimately, even with more optimized strategies it remains doubtful that it could compete with algorithms such as ASHC.

ASSRS, was a conceptually promising algorithm, that in the scope of this thesis failed to prove effectiveness. It showed high fail rates that effectively made the analysis of its competitiveness impossible. The main issue was that it relied on the basic concepts of local search without incorporating good mechanisms of escaping local optima. Refining these mechanisms to better and more reliably escape local optima, for example by making the worst possible move after getting stuck, could in future work make the algorithm much more competitive. The costly basic iteration, of moving all the way up the spanning tree for every single move however, would probably still limit that competitiveness quite heavily.

SHC, an algorithm that is also very close to basic local search, but introduces randomness in order to escape local optima, performed very well on all tested mesh sizes. It had high success rates, of over 93% and was faster than ASSRS and ILS by a large margin. It is a simple algorithm and still proved effective for computational unfolding. Making substantial changes to the SHC algorithm is unlikely in future work, apart from tweaking the probabilities of making bad steps or indifferent steps, to potentially further optimize its runtime.

ASHC emerged as the best algorithm in terms of both performance and success rate. It achieved a runtime decrease of 15.9% relative to regular SHC and maintained consistent success rates of 93% to 99%. ASHC added the concept of adapting the probability of making bad or indifferent moves to the SHC algorithm. Increasing the chances of such moves, the fewer overlaps remained. This led to ASHC minimizing wasted time in cases, where there were few or no options to further optimize the unfolding. Its consistent outperforming of all other implemented algorithms, shows the potential of adaptiveness in computational unfolding. In future work, optimizing the rate at which these probabilities change depending on overlaps could lead to a further increase in performance.

When comparing ASHC to the algorithms from the related works in Section 3, it can be observed that it outperforms simulated annealing and performs on a similar level to the genetic algorithm.

Ultimately, while ILS and ASSRS, in their current form, can not compete in the field of computational unfolding, SHC and especially ASHC can definitely be competitive. The findings of this thesis can serve as a foundation for developing the adaptive stochastic hill climbing algorithm further, to potentially address challenges in the future of 3D mesh unfolding.

# 9 Bibliography


[AEP20] Niclas Andréasson, Anton Evgrafov, and Michael Patriksson. *An introduction to continuous optimization: foundations and fundamental algorithms*. Courier Dover Publications, third edition, 2020.

[BT93] Dimitris Bertsimas and John Tsitsiklis. Simulated annealing. *Statistical Science*, 8(1):10–15, February 1993.

[BV04] Stephen P Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[DBS06] Marco Dorigo, Mauro Birattari, and Thomas Stützle. Ant colony optimization. *IEEE computational intelligence magazine*, 1(4):28–39, November 2006.

[GK03] Fred Glover and Gary A. Kochenberger. *Handbook of Metaheuristics*. Springer US, first edition, 2003.

[HE12] Thomas Haenselmann and Wolfgang Effelsberg. Optimal strategies for creating paper models from 3d objects. *Multimedia Systems*, 18(6):519–532, November 2012.

[HMR16] Lingaraj Haldurai, T. Madhubala, and R. Rajalakshmi. A study on genetic algorithm and its applications. *International Journal of Computer Sciences and Engineering*, 4(10):139–143, October 2016.

[JMT04] Hubertus Th. Jongen, Klaus Meer, and Eberhard Triesch. *Optimization Theory*. Springer Science & Business Media, 2004.

[KGV83] Scott Kirkpatrick, Daniel Gelatt, and Mario Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, May 1983.

[KTGW20] Thorsten Korpitsch, Shigeo Takahashi, Eduard Gröller, and Hsiang-Yun Wu. Simulated annealing to unfold 3d meshes and assign glue tabs. *Journal of WSCG*, 28(1–2):47–56, January 2020.

[MKA07] Wil Michiels, Jan Korst, and Emile Aarts. *Theoretical Aspects of Local Search*, volume 13. Springer Science & Business Media, 2007.

[MW15] Federico Marini and Beata Walczak. Particle swarm optimization (pso). a tutorial. *Chemometrics and Intelligent Laboratory Systems*, 149:153–165, August 2015.

[PR88] R. Gary Parker and Ronald L. Rardin. *Discrete optimization*. Academic Press, Inc., 1988.

[RLMS00] Helena Ramalhinho-Lourenço, Olivier C. Martin, and Thomas Stützle. Iterated local search. Economics Working Papers 513, Department of Economics and Business, Universitat Pompeu Fabra, November 2000.

[Rus21] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., global fourth edition, 2021.

[SS68] M. Schumer and K. Steiglitz. Adaptive step size random search. *IEEE Transactions on Automatic Control*, 13(3):270–276, 1968.

[TWS+11] Shigeo Takahashi, Hsiang-Yun Wu, Seow Hui Saw, Chun-Cheng Lin, and Hsu-Chun Yen. Optimized topological surgery for unfolding 3d meshes. *Computer Graphics Forum*, 30(7):2077–2086, November 2011.

# 10 Appendix

## 10.1 Pseudocode

**Listing 10.1: The simulated annealing algorithm.**

```
//Initialize algorithm
candidate_solution = generate_first_MST
temperature = max_temperature
cooling_rate = cooling_schedule

//Iterate while temperature is positive
while temperature > 0:

  //Create neighbor
  neighbor_solution = generate_neighbor(candidate_solution)

  //Calculate cost difference between new and old solution
  cost_delta = cost(neighbor_solution)-cost(candidate_solution)

  //Accept neighbor if better than candidate
  if cost_delta < 0:
    candidate_solution = neighbor_solution

  //Accept neighbor with a probability
  else:
    change_propability = exp(-cost_delta/temperature)

    if random() < change_probability:
      candidate_solution = neighbor_solution

  //Decrease temperature according to cooling schedule
  temperature = decrease_temp(cooling_schedule)
end
```

**Listing 10.2: The basic local search algorithm.**

```
// Takes a first potential solution
P = startingPoint()

// Iterates until the termination criterion is met
while !TerminationCriterion():

    // Finds the neighbors of P and selects the best one
    N = Neighbors(P)
    bN = BestNeighbor(N)

    // potential solution becomes the neighbor if its better
    if better(bN, P):
        P = bN

    // when theres no better neighbors, P is returned
    else:
        return P

// p is returned after meeting the termination criterion
return P
```

**Listing 10.3: The iterated local search algorithm.**

```
// Takes a first potential solution
P = startingPoint()

// Iterates until the termination criterion is met
while !TerminationCriterion():

    // Finds the neighbors of P and selects the best one
    N = Neighbors(P)
    bN = BestNeighbor(N)

    // potential solution becomes the neighbor if its better
    if better(bN, P):
        P = bN

    // when theres no better neighbors and the termination criterion isn't met, P is
       perturbed
    elif !TerminationCriterion():
        P = perturb(P);

// p is returned after meeting the termination criterion
return P
```

**Listing 10.4: The stochastic hill climbing algorithm.**

```
// Takes a first potential solution
P = startingPoint()

// Iterates until the termination criterion is met
while !TerminationCriterion():

    // Finds the neighbors of P and assigns them a probability according to their
        fitness
    N = Neighbors(P)
    Np = giveProbability(N)

    // Randomly chooses a neighbor with the given probability as the new candidate
        solution
    P = chooseNeighbor((N, Np))

// p is returned after meeting the termination criterion
return P
```

**Listing 10.5: The adaptive step size random search algorithm.**

```
// Takes a first potential solution and step size
P = startingPoint()
S = startingStepSize()
Ds = decreaseSchedule

// Iterates until the termination criterion is met
while !TerminationCriterion():

    // Finds neighbors of P at distance S and selects the best one
    N = Neighbors(P,S)
    bN = BestNeighbor(N)

    // potential solution becomes the neighbor if its better
    if better(bN, P):
        P = bN

    // when theres no better neighbors, P is returned
    else:
        return P

    // checks if its time to decrease step size and does so
    if stepSizeDecrease(Ds):
        S = decrease(S,Ds)

// p is returned after meeting the termination criterion
return P
```

## 10.2 Meshes

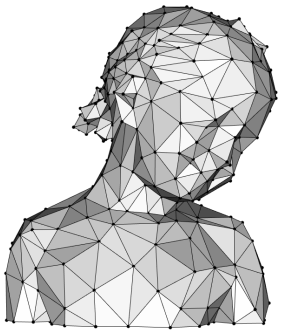


Figure 10.1: ”Bimba” 3D mesh

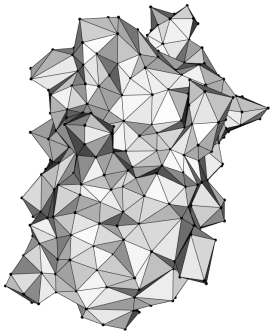


Figure 10.2: ”Blob” 3D mesh

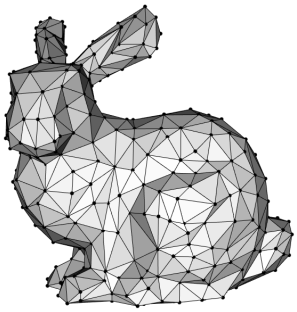


Figure 10.3: ”Bunny” 3D mesh

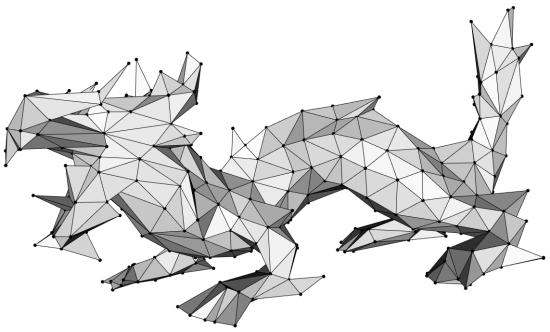


Figure 10.4: ”Dragon” 3D mesh

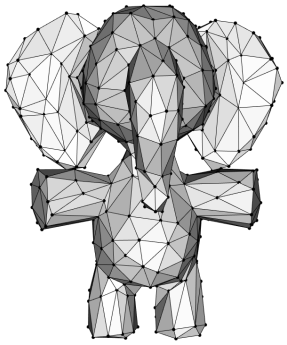


Figure 10.5: ”Elephant” 3D mesh

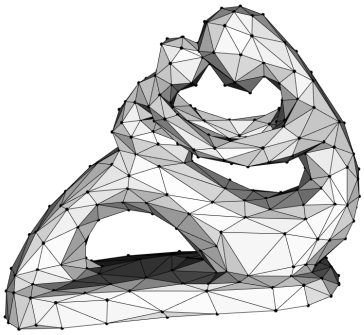


Figure 10.6: ”Fertility” 3D mesh

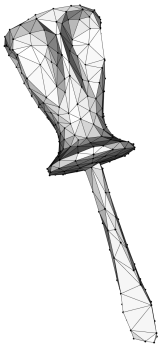


Figure 10.7: ”Screwdriver” 3D mesh

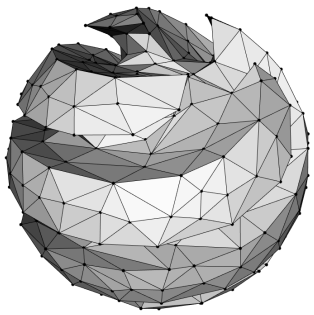


Figure 10.8: ”Sharp_sphere” 3D mesh

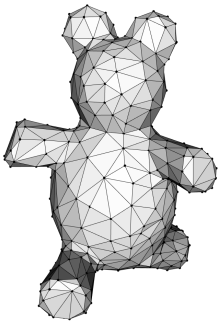


Figure 10.9: ”Teddy” 3D mesh

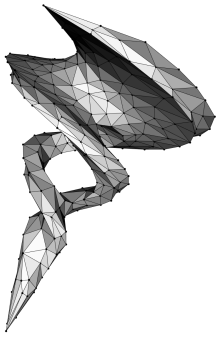


Figure 10.10: ”Twirl” 3D mesh