DOI: 10.1111/cgf.142632
EUROGRAPHICS 2021 / N. Mitra and I. Viola
(Guest Editors)

Volume 40 (2021), Number 2

Layout Embedding via Combinatorial Optimization

J. Born[1] and P. Schmidt[1] and L. Kobbelt[1]

[1]RWTH Aachen University, Germany

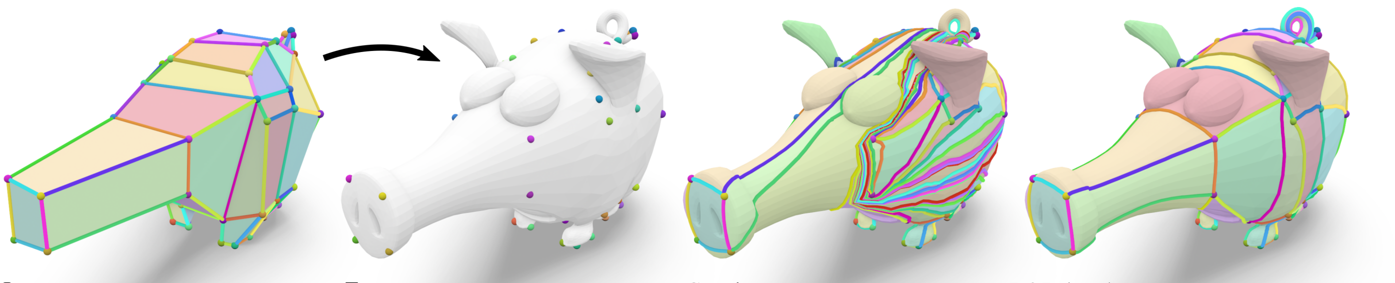


Figure 1: *We embed a given layout connectivity (left, visualized as a coarse mesh for illustration) into a target surface with prescribed landmark positions (center left) by successively embedding edges as shortest paths in an optimized order. Previous methods choose a greedy sequence of locally optimal decisions, which can lead to severe topological artifacts (center right) due to accidental blocking of subsequently inserted paths. Our branch-and-bound method globally optimizes over all possible edge insertion sequences to find an embedding of shortest total path length, which yields the expected homotopy class (right).*

Abstract

We consider the problem of injectively embedding a given graph connectivity (a layout) into a target surface. Starting from prescribed positions of layout vertices, the task is to embed all layout edges as intersection-free paths on the surface. Besides merely geometric choices (the shape of paths) this problem is especially challenging due to its topological degrees of freedom (how to route paths around layout vertices). The problem is typically addressed through a sequence of shortest path insertions, ordered by a greedy heuristic. Such insertion sequences are not guaranteed to be optimal: Early path insertions can potentially force later paths into unexpected homotopy classes. We show how common greedy methods can easily produce embeddings of dramatically bad quality, rendering such methods unsuitable for automatic processing pipelines. Instead, we strive to find the optimal order of insertions, i.e. the one that minimizes the total path length of the embedding. We demonstrate that, despite the vast combinatorial solution space, this problem can be effectively solved on simply-connected domains via a custom-tailored branch-and-bound strategy. This enables directly using the resulting embeddings in downstream applications which cannot recover from initializations in a wrong homotopy class. We demonstrate the robustness of our method on a shape dataset by embedding a common template layout per category, and show applications in quad meshing and inter-surface mapping.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Computer graphics;*** *Mesh geometry models; Mesh models;*

1. Introduction

Many mesh processing methods decompose an input object into an arrangement of patches by cutting along paths on the surface, giving rise to an embedded cell complex. The structure of this cell complex, the *layout*, is often derived from the input geometry in an automatic generation process. In this work, we consider the reverse problem of embedding a *prescribed* layout into a given target surface.

The need to prescribe a specific layout arises in many applications: A layout itself may be used to assign semantic information of a general structure (e.g. annotations on parts of a design template or skeleton) which is then carried over to particular instances. For shape collections, a common layout may act as a base domain to

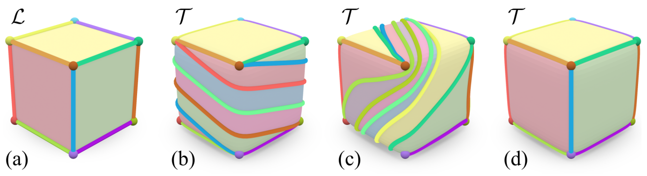


Figure 2: *When landmark positions are fixed, there are infinitely many topologically distinct ways to embed a layout (a) into a target surface (b, c, d). We are interested in finding embeddings of shortest total path length (d) since they correspond intuitively to the correct embedding.*

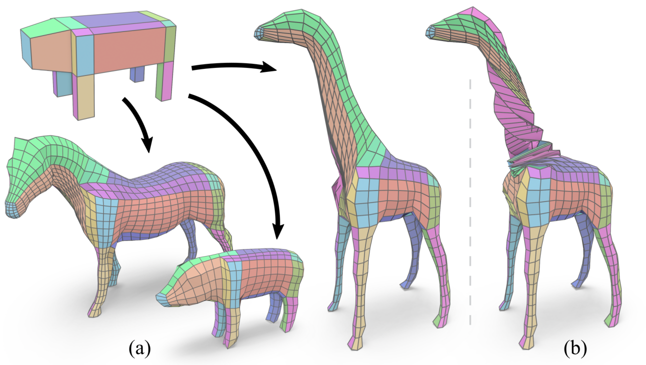


Figure 3: *One application scenario which relies on robust layout embedding is quad meshing with prescribed base complex. Results heavily depend on finding embeddings in the correct homotopy class (a), thus avoiding unintended twists (b).*

define shape correspondence across different members. When the collection is extended by additional models, the shared layout template needs to be embedded into each new object. For specific processing tasks, such as re-meshing, a layout directly controls certain aspects of the outcome, e.g. the resulting base complex of a tessellation (Fig. 3). In these cases, it can be desirable to prescribe a (hand-crafted or previously generated) layout, allowing to explicitly model structural features (such as symmetries or repetitions) or standardized functional components (e.g. connectors or boreholes in technical parts) in a controlled way.

In the following, we therefore consider the task of automatically embedding a given layout (defined by a coarse cell complex without geometry) into a target surface. We assume that the target positions for layout vertices are provided as landmarks on the input surface. The problem of layout embedding then amounts to finding a non-intersecting network of embedded paths on the surface, conforming to the structure of the layout.

Maybe surprisingly, this problem has a vast space of possible solutions: While a layout only mandates *which* points should be connected via paths, an embedding must specify *how* these paths are routed across the surface. Certainly, every path has a continuum of possible *geometric* shapes. But even if we ignore continuous deformations of paths and only look at their homotopy, there are infinitely many *topological* choices how to route a path around the other embedded layout vertices (Fig. 2).

Faced with this enormous set of choices, it is natural to state a preference for embeddings that are in some sense simplest or shortest (e.g. by total geodesic length of all embedded edges). We argue that shortest embeddings tend to match human intuition of the "correct" topology as they avoid unnecessary twists and detours.

To simplify the task of finding *shortest* layout embeddings, we restrict to a certain class of embeddings that is only parametrized by discrete degrees of freedom and eliminates continuous choices: We consider *shortest-path embeddings*, which are constructed by successively embedding layout edges in a certain order as (non-intersecting) shortest paths. All such embeddings are uniquely identified by an insertion sequence of edges, which implicitly encodes topological decisions: Depending on the ordering, shortest edge embeddings naturally assume certain homotopy classes by avoiding intersections with earlier insertions (Fig. 4).

Shortest-path embeddings have been employed in numerous works for the embedding of layouts [PSS01] or cut graphs [KSG03, KS04, SAPH04, APL14]. All previous works rely on greedy strategies for picking an insertion sequence, basing their decisions on local path lengths and several heuristics. However, these methods allow no backtracking: When heuristics fail, wrong choices in earlier stages can lead to spectacularly bad embeddings later on, exhibiting paths in unexpected homotopy classes (and of excessive length) that swirl around remote regions of the target surface (cf. Fig. 1).

Downstream applications that use embeddings for further processing (e.g. global chart-based parametrization, re-meshing, or inter-surface mapping) can typically not recover from initializations in a wrong homotopy class [SAPH04, APL15, SCBK20]. Due to their weak reliability, greedy methods for layout embedding require human supervision and intervention, making them poorly suited for automatic shape processing pipelines.

In this work, we go beyond the greedy paradigm and present a method that systematically searches for optimal insertion sequences yielding shortest-path embeddings of minimal total length. Our search is powered by a custom branch-and-bound strategy,

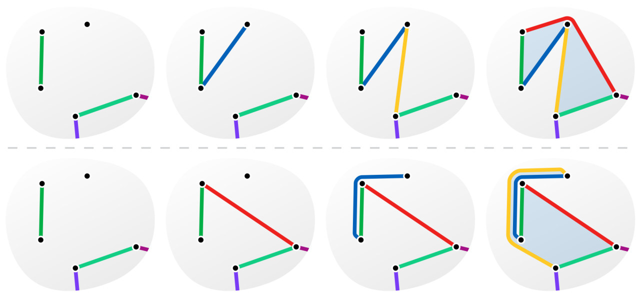


Figure 4: *Starting from a partial embedding (left), three additional edges are successively inserted as shortest paths. Depending on the insertion order (top row: blue, yellow, red), (bottom row: red, blue, yellow), we get different embeddings with paths and patches (light blue) in different homotopy classes.*

crawling the decision tree that governs the incremental construction of embeddings. We demonstrate that—despite the vast search space—it is possible to quickly and reliably find low-cost solutions by exploiting domain knowledge in the form of bounding and pruning rules and a specialized search priority. We can incorporate previous greedy strategies into this system by using their results as initial solutions, which are then improved or confirmed by our algorithm.

The branch-and-bound framework affords the additional flexibility to tune this method to a desired quality or performance: Since results are accompanied by quality bounds, it is possible to solve the problem up to a certain optimality tolerance. When running the algorithm on a fixed time budget, it terminates with an optimality estimate. Besides these (optional) performance specifications, our algorithm is parameter-free.

1.1. Contribution

We make the following contributions:

- a method for embedding a given layout into a simply-connected target surface (with prescribed vertex positions) while explicitly optimizing for minimum embedding path length,
- a combinatorial formulation of this problem in terms of edge insertion sequences,
- a custom branch-and-bound strategy, tailored to this task.

We evaluate the performance of our method against several greedy heuristics and demonstrate applications in quad meshing and inter-surface mapping. Along with this publication, we release the source code of our implementation as supplementary material.

2. Related Work

We briefly review works in a number of fields adjacent to our problem setting.

Layout Generation There exists a multitude of methods that generate coarse layouts for given models. Different approaches produce triangular [EH96, LSS*98, KLS03, SAPH04] or quadrilateral layouts [EH96, DBG*06, CBK12, BCE*13, RRP15, ULP*15, SBLS18] (just to name a few; see [Cam17] for an extensive survey). While the above methods generate layouts in tandem with an embedding into the input surface, they can also be applied for the sole purpose of extracting layout templates. Besides fully automatic methods, interactive user-guided layout generation tools [TPSHSH13, CK14a] have been proposed as well.

Computational Topology Various (shortest-path) surface decomposition problems have been studied. Among them are shortest cut graphs (NP-hard in general [EHP04], but an efficient algorithm exists if vertices are prescribed [CdV10]), shortest homotopy bases [EW05], and canonical polygon schemata [Liv20]. Shortest embeddings of a *fixed* connectivity are of interest in graph drawing: [LMM*95] consider the problem of finding shortest non-intersecting paths between pairs of points in the plane (later proven to be NP-hard [BF98]). [CHKL13] present polynomial-time approximation algorithms for shortest embeddings of general planar graphs. Our layout embedding problem is more constrained: In addition to the graph structure of a layout, we also prescribe an ordering of edges around each vertex. From a topological perspective, the problem of layout embedding amounts to finding the homotopy class of an injective map from the layout to the target surface. Algebraically, the potential solutions are precisely the elements of the pure mapping class group [FM11] of the target surface punctured at the layout vertices.

Greedy Shortest-Path Embeddings Layout embedding problems appear in the literature in various formulations. In a problem setting identical to ours, [PSS01] also embed a given layout into a target surface, but instead of systematically searching for an optimal insertion sequence, the embedding is created based on a series of local decisions, guided by heuristics. Similar greedy approaches are employed for slightly different surface decomposition tasks, where no layout connectivity is prescribed: the compatible triangulation of landmark vertices on pairs of meshes [KSG03, KS04, SAPH04], or the simultaneous computation of matching cut graphs [APL14, APL15]. The above works employ hand-crafted combinations of heuristics to guide their decisions. We adapt those from [PSS01, KSG03, SAPH04] to our setting and evaluate their performance in Sec. 5.1. While we demonstrate that these heuristics can fail in certain situations, they are still valuable for our method as their results can serve as initial solutions for our branch-and-bound search.

Homotopy-Preserving Embedding Optimization Our method aims to find the most natural embedding topology by optimizing *across* different homotopy classes. Various ways to optimize embedding geometry *within* a given homotopy class have been proposed, which can be applied as a post-process for our results: Path-based methods include active models that continuously evolve an explicit curve representation [LL02, BWK05, YSC20] or discrete methods for finding exact shortest homotopic geodesics [HS94, SC20]. Other methods consider the interior of patches and optimize embeddings based on mapping distortion [KS04, SAPH04, SCBK20] or curvature alignment [CK14b]. Another simple approach is re-tracing paths as straight lines in local planar parametrizations [PSS01].

Branch-and-Bound Optimization A number of general-purpose numerical solvers for (mixed-)integer linear, quadratic, or nonlinear optimization use branch-and-bound as a core component [MJSS16]. Off-the-shelf solvers (e.g. [CPL09, GO20, GAB*20]) usually require an explicit model description (often with certain restrictions on the objective function). Our problem, involving sequences of constrained shortest-path cost computations cannot be naturally encoded in such a framework. Instead, we rely on a simple custom branch-and-bound implementation specifically tailored to the objective and constraints of the layout embedding problem.

3. Layout Embedding

A **layout** $\mathcal{L}$ is an abstract 2-dimensional cell complex: a purely combinatorial description of a 2-manifold polygonal mesh, carrying no geometric information. Its connectivity is defined by a simple, connected graph $(V_{\mathcal{L}}, E_{\mathcal{L}})$. Additionally, a layout specifies

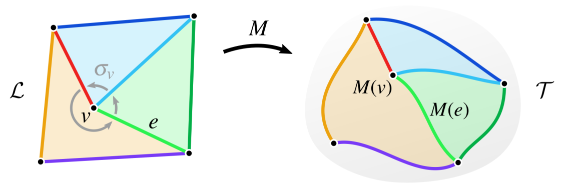


Figure 5: *Vertices and edges of a layout $\mathcal{L}$ are mapped to points and paths on a target surface $\mathcal{T}$ via an embedding M. Embeddings are injective and preserve the cyclic ordering σ_v of edges around each layout vertex v.*

cyclic orderings σ_v of edges around each vertex v, which imply a set of layout faces and their adjacency.

We consider a **target surface** $\mathcal{T}$: an orientable, simply-connected (genus 0 or disk) 2-manifold surface.

An **embedding** $M : \mathcal{L} \to \mathcal{T}$ is an injective map that assigns layout vertices and edges to points and paths on the target surface $\mathcal{T}$ (Fig. 5). Specifically:

- Every vertex $v \in V_{\mathcal{L}}$ is mapped to a distinct point $M(v) \in \mathcal{T}$.
- Every edge $e = (v_1, v_2) \in E_{\mathcal{L}}$ is embedded as a continuous path $M(e)$ on $\mathcal{T}$ with $M(v_1)$ and $M(v_2)$ as endpoints. Embedded edges do not (self-)intersect and only touch embedded vertices at their endpoints.
- The cyclic order of incident paths around each embedded vertex $M(v)$ respects the order σ_v defined by the layout.

We also consider *partial* embeddings where only a subset of layout edges is mapped. We denote this subset of *embedded edges* by $E(M) \subseteq E_{\mathcal{L}}$ and the set of *unembedded edges* as its complement $\overline{E}(M) = E_{\mathcal{L}} \setminus E(M)$. A *complete* embedding cuts the target surface into patches which correspond to the faces implied by the layout.

3.1. Shortest-Path Embeddings

Given a partial embedding M and an unembedded layout edge $e \in \overline{E}(M)$, we define $p(M,e)$ as the **shortest path** on $\mathcal{T}$ that yields a valid extension of M. Such a path does not intersect already embedded edges and respects the cyclic ordering of incident embedded edges at its endpoints.

An **insertion sequence** $s = e_1 \cdots e_n \in E^*_{\mathcal{L}}$ is a (partial) permutation of layout edges. It describes the construction of an embedding via successive addition of edges embedded as shortest paths: Starting from an "empty" embedding M_0 that only maps vertices, we successively construct embeddings $M_1, \ldots, M_n$, corresponding to the elements of s. Each M_k is identical to the previous M_{k-1}, but additionally includes an embedding of e_k, defined as $M_k(e_k) = p(M_{k-1}, e_k)$. The final embedding M_n includes all edges in s. We call it the **shortest-path embedding** of s, denoted by $M(s)$.

Note that different insertion sequences of the same set of layout edges can result in different shortest-path embeddings (Fig. 4): In general, $M(s) \neq M(\pi(s))$ for a permutation π.

We define the cost $c(M(e))$ of a path as its length on $\mathcal{T}$. The cost $c(M)$ of a (partial) embedding is the sum of costs of all embedded edges. It can happen that in an embedding M, no shortest path $p(M,e)$ exists for some edge e due to blocking by other paths. Consequently, it is possible that an insertion sequence s does not imply a valid shortest-path embedding. We consider such invalid paths and embeddings to have infinite cost.

3.2. Problem Statement

We address the following problem: Given a layout $\mathcal{L}$, a target surface $\mathcal{T}$, and an initial embedding M_0 that assigns target points for all layout vertices, find the insertion sequence s that produces a shortest-path embedding $M(s)$ of minimum cost:

$$\underset{s \in E^*_{\mathcal{L}}}{\operatorname{argmin}}\ c(M(s))$$

3.3. Discrete Representation

In practice, we use discrete representations of target surfaces and embeddings. A target surface $\mathcal{T}$ is given by a triangle mesh $(V_{\mathcal{T}}, E_{\mathcal{T}}, F_{\mathcal{T}})$. We represent an embedding by mapping directly to mesh elements: Each layout vertex $v \in V_{\mathcal{L}}$ maps to a target vertex $M(v) \in V_{\mathcal{T}}$. Each layout edge $e \in E_{\mathcal{L}}$ maps to a chain of target edges $M(e) \subset E_{\mathcal{T}}$, connecting the corresponding endpoints.

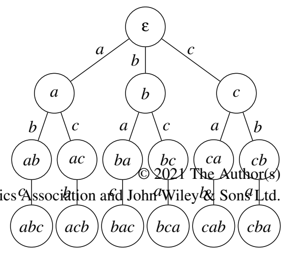
In this representation, shortest paths $p(M,e)$ can be found via Dijkstra's algorithm. As described in several prior works (e.g. [PSS01]), the search is constrained to maintain injectivity of the embedding: New paths must not cross or touch previously inserted edges. Additionally, when tracing paths between vertices with incident embedded edges, those vertices may only be approached from sectors that are consistent with the cyclic orderings σ_v defined by the layout.

Following [SAPH04] and others, we adaptively refine the target mesh to accommodate paths in mesh regions that are not sufficiently tessellated: During path tracing, we allow paths to tentatively travel across edge midpoints of the mesh $\mathcal{T}$ (Fig. 6 (a)). When a path is added to an embedding, we insert the required edge midpoints into $\mathcal{T}$ via local edge splits (Fig. 6 (b)).

As a consequence of this discrete representation, resulting shortest paths can appear slightly jagged (due to the tessellation of the underlying mesh). A variety of (homotopy-preserving) post-processes can be applied to remove such discretization artifacts and improve the shape of embeddings (cf. Sec. 2). We use the path smoothing method described by [PSS01] (retracing each edge path in a harmonic parametrization of the incident faces, Fig. 6 (c)) in our visualizations.

4. Branch-and-Bound Optimization

The incremental construction of a layout embedding via successive shortest path insertions consists of a sequence of decisions: Which layout edge should be embedded next? We can describe this process by a decision tree in which nodes represent intermediate partial embedding states and each edge indicates an insertion decision. Each node is uniquely identified by an insertion sequence s, encoding the (partial) embedding $M(s)$. (In the following, we will use the terms



node, *embedding state*, and *insertion sequence* interchangeably.) The root node is the empty insertion sequence ε, corresponding to the initial state where only vertex correspondences are given and no edges are embedded. The outgoing edges of each interior node s represent possible extensions of the partial embedding $M(s)$: Each unembedded layout edge $e \in \overline{E}(M(s))$ is a candidate for insertion, leading to a child node identified by the sequence se. Therefore, every node s is a prefix of all of its descendants s', we write $s \sqsubseteq s'$. The $|E_{\mathcal{L}}|!$ leaf nodes of the tree correspond to full embedding sequences, which are potential solutions to our problem (cf. Sec. 3.2).

Already for layouts with a moderate number of edges, the factorial size of this decision tree makes an exhaustive search for the optimal solution infeasible. In the following, we describe our branch-and-bound strategy that finds solutions while only exploring a fraction of the search tree. Effective bounding allows our method to prune the search space and to precisely judge the optimality of the solutions it discovers.

4.1. Algorithm

Essentially, the branch-and-bound algorithm is a tree search that crawls the decision tree along a front propagating from the root. During the entire search, it keeps track of an **incumbent solution** s^*: an insertion sequence representing the best (i.e., lowest-cost) solution encountered so far. The cost of the current incumbent defines a **global upper bound** $c^\top = c(M(s^*))$. At any time, this upper bound is greater than (or equal to) the cost of the true optimal solution of the problem. It decreases monotonically with each update, approaching the optimum.

We can use greedy heuristic methods (cf. Sec. 5 for details) to produce an initial incumbent and upper bound. If no heuristic initialization is used, we start with $c^\top = \infty$.

Besides global upper bound estimates, a vital element of any branch-and-bound method is the ability to additionally compute lower bound cost estimates for individual states of the decision tree. For each state of the tree, identified by a partial insertion sequence s, we define a **local lower bound** $c_\perp(s)$ fulfilling the following property: Any state s' in the subtree rooted at s yields an embedding with a cost of at least $c_\perp(s)$, i.e.

$$c_\perp(s) \leq c(M(s')) \quad \forall s \sqsubseteq s'.$$

Lower bounds are typically obtained via a relaxation of the remaining sub-problem. We explain the computation of lower bounds in our setting in Sec. 4.3.

Available information on upper and lower bounds can be exploited to prune the decision tree. Any time a state s has a lower bound $c_\perp(s)$ that exceeds the current global upper bound $c^\top$, we can safely ignore it (and its entire subtree) in our search: By definition of $c_\perp(s)$, all solutions in the subtree of s will be more costly than $c^\top$ and therefore cannot improve the incumbent s^*.

The order in which states (nodes of the search tree) are visited is controlled by a function that assigns a priority $P(s)$ to any state s. We discuss our design of the priority function $P(s)$ in Sec. 4.5. Potential states to be visited are stored in a **priority queue** Q sorted by P. Initially, this queue only contains the root node, i.e. $Q = \{\varepsilon\}$.

The core of our branch-and-bound algorithm is a loop that consumes the priority queue Q. In each iteration, the state s with the highest priority is extracted from Q and processed as follows:

1. **Update**: Whenever the insertion sequence s yields a complete embedding $M(s)$, it is a potential solution (i.e., a leaf node of the decision tree). If its cost $c(M(s))$ improves the current incumbent solution s^*, we record s as the new incumbent and update the global cost upper bound $c^\top$ accordingly.
2. **Branch**: If s corresponds to a partial embedding, we enumerate all states that can be reached from s by inserting an additional layout edge $e \in \overline{E}(M(s))$ (child nodes of s in the decision tree) and add them to Q for future exploration.
3. **Bound**: For each new state s' that is added to Q, we perform a bounds check: If the cost lower bound $c_\perp(s')$ is larger than the current global upper bound $c^\top$, it can be pruned: Instead of adding s' to Q, we simply discard it.

A typical behavior for Q is to initially keep growing until an upper bound $c^\top$ is reached that facilitates enough pruning to reverse the growth of Q. The algorithm terminates when Q is exhausted.

4.2. Optimality Gap

At any time during the algorithm, Q contains a set of states along the current propagation front of the tree search. By examining the lower bounds of states in Q, we can compute a **global lower bound**

$$c_\perp = \min_{s \in Q} c_\perp(s)$$

which bounds the cost of all unseen solutions further down the tree. The difference between the two global bounds $c^\top - c_\perp$, called the **optimality gap**, quantifies by how much the incumbent solution could possibly improve during the remainder of the search. Fig. 7 visualizes how the optimality gap gradually closes as global upper and lower bounds converge over the course of an optimization.

When the algorithm terminates, we know the incumbent s^* is a global optimum. Instead of waiting for termination, the algorithm

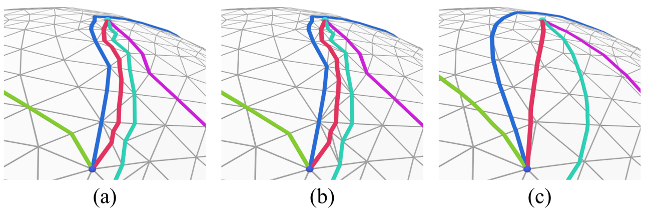


Figure 6: *A new edge embedding (red) is computed as a shortest intersection-free path on $\mathcal{T}$. During tracing (a), the path may travel across edge midpoints. Upon insertion (b), midpoints are incorporated into $\mathcal{T}$ via local refinement. After the embedding is complete, we apply path smoothing to the entire embedding (c).*

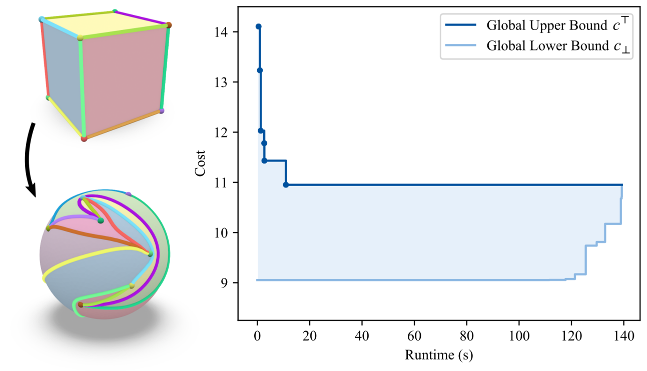


Figure 7: *Progression of global upper bound $c^\top$ and global lower bound $c_\perp$ over the course of an optimization (here: embedding a cube layout into a sphere with random target vertex positions). The optimal solution is already found after 11 s. Optimality is proven after 139 s.*

can also be stopped prematurely. In that case, s^* may not be optimal, but we can give an estimate of its quality: We know that if there is a better solution, its cost could at most be better by the current optimality gap. This mechanism gives us the flexibility to run the branch-and-bound algorithm on a time limit.

Likewise, we can choose to terminate the algorithm early if the current optimality gap falls below an acceptable threshold. When the relative optimality gap $\frac{c^\top - c_\perp}{c^\top}$ falls below a certain tolerance fraction α, we stop execution. The same tolerance can be applied for all pruning operations, which further reduces the search space. We use $\alpha = 1\,\%$ in all of our experiments.

In the following, we discuss customizations of this general branch-and-bound setup, which are specifically tailored to the structure of our layout embedding problem.

4.3. Lower Bounds

Given a state s, corresponding to a partial embedding $M(s)$, we want to compute $c_\perp(s)$: A lower bound on the cost of any full embedding $M(s')$ that can be reached from s by embedding all remaining layout edges $\overline{E}(M(s))$.

One valid lower bound is the cost of the current partial embedding $c(M(s))$ (Fig. 8 (a)): All edges already embedded in $M(s)$ are identical in any extension $M(s')$ and the insertion of the remaining edges in $M(s')$ will only incur additional cost. By itself, this lower bound is fairly ineffective because it entirely ignores the potential cost of embedding the remaining part of the layout.

We can obtain a tighter bound by also simulating the insertion cost of the remaining edges in a setting that relaxes the constraints on a valid embedding: For each unembedded edge $e \in \overline{E}(M(s))$, we compute a **candidate path**, i.e. its shortest-path embedding $p(M(s), e)$. While each candidate by itself is compatible with the current state $M(s)$, the set of all candidates will likely not constitute a valid embedding (Fig. 8 (b)): It is possible that candidates mutually intersect or violate each other's cyclic ordering requirements. Still, due to each candidate path being individually shortest, we know that any modification to resolve these conflicts would only increase cost (Fig. 8 (c)). Therefore, the sum of candidate path costs serves as a lower bound for the cost of any valid completion. In combination with the cost of already embedded edges, we arrive at the following cost lower bound for a state s:

$$c_\perp(s) = c(M(s)) + \sum_{e \in \overline{E}(M(s))} c(p(M(s), e)).$$

By definition (Sec. 3.1), we consider path costs to be infinite where no valid path exists. In a situation where one of the candidate paths has no valid embedding (due to being blocked by different, already inserted paths), the lower bound $c_\perp(s)$ becomes infinite, thereby identifying the current state as a "dead end" and excluding it from further processing.

4.4. Pruning Redundant Subtrees

Besides pruning based on upper and lower bounds, we employ specialized pruning rules to reduce redundant computations in different branches of the decision tree.

4.4.1. Detecting Redundant States

It is possible that two different insertion sequences s_1, s_2 lead to an identical (partial) embedding $M(s_1) = M(s_2)$. In that case, the entire subtrees of s_1 and s_2 are identical as well. If we have already visited s_1 (or know that we will visit it), we can safely ignore s_2 and its entire subtree in our search.

Whenever we consider adding a new state s to the priority queue Q for exploration, we compute a lightweight hash of its embedding $h(M(s))$ and look it up in a table H of known embeddings. If $h(M(s))$ is already in H, we simply discard s. Otherwise, we add s to Q and insert its hash $h(M(s))$ into H.

In our discrete setting, an embedding hash $h(M)$ can be computed as follows: For each layout edge $e \in E_{\mathcal{L}}$, we form a string w_e by concatenating the vertex positions along the corresponding embedded edge $M(e)$. If e is not embedded in M, we use a blank

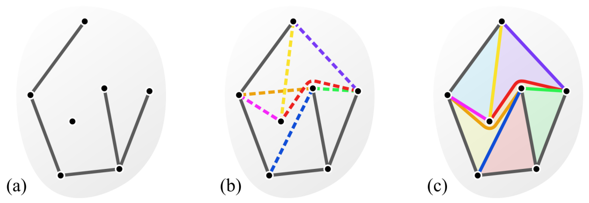


Figure 8: *We compute a lower bound for the cost of any valid completion of a partial embedding (a): Besides measuring the length of embedded edges (solid gray lines), we compute candidate paths (dashed lines) for unembedded layout edges (b). These are compatible with the partial embedding but possibly mutually conflicting. The sum of these path lengths is a lower bound for the cost of any embedding where all conflicts are resolved (c).*

symbol $w_e = \#$. We concatenate all strings w_e in a canonical order and compute a hash of the resulting word to obtain $h(M)$.

The above strategy detects redundant states when different branches of the decision tree have arrived at an identical embedding. In the following, we describe additional rules that can predict when the exploration of certain subtrees will only lead to redundant results so we can prune proactively.

4.4.2. Delaying Non-Conflicting Insertions

In a state s, each unembedded edge $e \in \overline{E}(M(s))$ is associated with a candidate path $p(M(s), e)$. Inserting an edge e will lead to a new state in which other edges have different candidate paths in general. However, there are some edges (called **non-conflicting edges**) whose insertion will not change the candidate paths of any other edges. When inserting only non-conflicting edges, the outcome will hence *not* depend on their order: For a set of n non-conflicting edges, all of their $n!$ insertion sequences will lead to an identical embedding. We detect non-conflicting edges and avoid the redundant computation of exploring all their permutations and thus significantly reduce the effective branching degree of our search tree.

A non-conflicting set of unembedded edges $\overline{C} \subseteq \overline{E}(M(s))$ is a set for which all insertion sequences lead to the same embedding; i.e. $M(s\pi(\overline{C}))$ is identical for any permutation π. For each state, we define $\overline{C}(s)$ as the maximal non-conflicting set and its complement, the set of conflicting edges, as $C(s) = \overline{E}(M(s)) \setminus \overline{C}(s)$. We consider the following two cases:

- If $C(s)$ is empty, all remaining unembedded edges are the non-conflicting edges $\overline{C}(s)$. In this case, the remaining edges can be inserted in any order, all leading to the same solution. Instead of exploring further child states, we immediately insert the remaining edges in an arbitrary order and terminate the search in this branch.
- If $C(s)$ is not empty, there are insertion decisions that matter: Every insertion of an edge from $C(s)$ has consequences for the embedding of at least one other edge, so we must consider these states in our search. In contrast, insertions of edges from $\overline{C}(s)$ have no immediate effect on other edges, so inserting them at this point would only introduce redundant branching. We therefore skip all insertions from $\overline{C}(s)$. Essentially, this delays the insertion of non-conflicting edges until they either become conflicting in some later state (at which point they are considered as an insertion option), or until only non-conflicting edges remain, which are then inserted simultaneously (see case above).

(We prove that this ruleset indeed only excludes redundant states in Appendix A.)

An extreme effect of this pruning strategy can be observed when the initial network of candidate paths at the root state ε is already conflict-free (which can in fact happen on simple inputs). In that case, $C(\varepsilon)$ is empty and our algorithm immediately terminates, returning the optimal result: An arbitrary insertion sequence of all edges.

Detecting Non-Conflicting Edges For each unembedded edge $e_i \in \overline{E}(M(s))$, we have already computed a candidate path $p_i = p(M(s), e_i)$ as part of the lower bound estimate of state s (Sec 4.3).

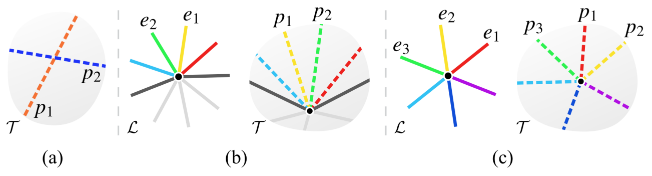


Figure 9: *There are three types of conflicts between candidate paths (dashed lines): (a) path intersections, (b) wrong path ordering within a sector created by already embedded edges (gray lines), (c) wrong path ordering around a vertex without sectors.*

We can determine the sets of conflicting and non-conflicting edges $C(s)$, $\overline{C}(s)$ based on the current configuration of candidate paths p_i:

Clearly, if two candidate paths p_1 and p_2 intersect (Fig. 9 (a)), the edges e_1 and e_2 are in conflict and we can add them to $C(s)$.

Additional conflicts can arise from the cyclic ordering requirements of paths around embedded vertices (Sec. 3): A path p_i may affect incident candidate paths at its endpoints, requiring them to approach the vertex from a different direction if p_i were inserted earlier. We detect such conflicts by verifying the current ordering of candidate paths around each embedded vertex $M(v)$. We distinguish two cases:

- If the layout vertex v has incident edges that are already embedded, the corresponding outgoing paths divide the region around $M(v)$ into sectors (Fig. 9 (b)). All candidate paths incident to $M(v)$ are already in their correct sector (otherwise their embedding would be invalid), but their relative ordering within a sector may be incompatible with the ordering σ_v imposed by the layout. In this case, it suffices to check conflicts based on a linear ordering: If an edge e_1 comes before another edge e_2 (in a counterclockwise sense) within a layout sector, the candidate path p_1 must come before p_2 in the embedded sector. Otherwise, e_1 and e_2 are conflicting and are added to $C(s)$.
- If $M(v)$ has no incident embedded edges, there are no sectors, so we need to verify the cyclic ordering of all candidate paths (Fig. 9 (c)): For every triplet $(e_1, e_2, e_3) \in \sigma_v$ of layout edges around v (in counterclockwise order), the corresponding candidate paths p_1, p_2, p_3 must have the same cyclic ordering at $M(v)$ in the embedding. Otherwise, we consider the edges e_1, e_2, e_3 as conflicting and add them to $C(s)$.

4.5. Priority

A common strategy to explore the search tree is to prioritize states with a small cost lower bound $c_{\perp}(s)$, motivated by the expectation to potentially find the lowest-cost solution there. In our observations, this strategy performs poorly: During early stages of the search, lower bounds can be quite similar across many states. With lower bounds monotonically increasing along branches of the decision tree, this priority leads to an approximate breadth-first traversal where large portions of the tree are expanded before reaching any leaf nodes that produce potential solutions. Since significant pruning of states can only happen after suitable upper bounds have been found, this search strategy essentially explores almost the entire decision tree, which is practically impossible.

We suggest a different exploration priority that aims to produce upper bounds early on (to facilitate pruning) while still steering towards states with promising lower bounds (to find potentially better solutions). The idea is to favor the pursuit of branches that have already made significant progress towards a solution.

For a given state s, we quantify this progress as follows: After computing candidate paths for the unembedded edges of $M(s)$, we apply the classification into conflicting and non-conflicting edges $C(s)$ and $\overline{C}(s)$ (Sec. 4.4.2). We use the number of conflicting edges $|C(s)|$ to judge how far s is from a complete solution: Starting from s, we estimate that $|C(s)|$ decisions are needed to resolve all remaining conflicts. We combine this information with the cost lower bound of s to define

$$P(s) = |C(s)| \cdot c_{\perp}(s)$$

and prioritize states where $P(s)$ is smallest.

4.6. Implementation Notes

During optimization, we need to represent different partial embeddings on the target triangle mesh $\mathcal{T}$. In each iteration, we extract a state s from the queue Q and reconstruct the embedding $M(s)$ by starting from the original mesh $\mathcal{T}$ and inserting the sequence s, locally refining the mesh as needed (cf. Sec. 3.3).

During our search, we build a lightweight representation of the decision tree explored so far, which allows us to cache intermediate results and avoid costly recomputations of paths. Whenever we produce a new state se from a parent state s, we save the newly computed path $p(M(s), e)$ at the corresponding edge of the decision tree. We can later reconstruct the embedding $M(s)$ by collecting all cached paths along the branch from ε to s and inserting them in that order.

The computation of candidate paths (Sec. 4.3) can be cached in a similar way: Initially, we compute candidate paths for all edges independently and store them in the root node ε. For all subsequent states, we only need to recompute candidate paths for the edges that are affected by a new insertion. In each child state se, those are precisely the edges that were in conflict with e in the parent state s (Sec. 4.4.2). It suffices to only cache the set of updated candidate paths in each state: To reconstruct the full set of candidate paths, we follow the tree towards its root until a cached path is found for all edges.

In all our examples, the total memory consumed by the queue Q (Sec. 4.1), the hash table H (Sec. 4.4.1), and the hierarchical cache of shortest paths never exceeded 188 MB.

5. Results and Applications

In the following we evaluate the performance of our method, compare against greedy approaches with different heuristics, and demonstrate application scenarios that benefit from the robustness of our method.

Greedy Ordering Heuristics We compare against three different greedy methods that base their decisions (which edge to embed next) on the heuristics that appear in [PSS01, KSG03, SAPH04]. While [PSS01] addresses our exact problem setting, we adapt the ideas presented in [KSG03] and [SAPH04] from the *variable*-layout setting to our *fixed*-layout setting:

- [PSS01] use a "swirl detection" heuristic to delay path insertions that are likely to cause topological artifacts: Edge embeddings are postponed if opposite layout vertices of the inserted edge are located on the wrong side of the tentative path, based on a proximity check. Further, embedded paths are traced using a custom metric that pushes paths away from landmark vertices. To prevent dead ends, insertions that close a cycle of edges are delayed until a spanning tree of the entire layout is embedded.
- Instead of relying on a conservative spanning tree heuristic, [KSG03] (and [KS04]) only prevent insertions that actually lead to a blocking of future paths.
- [SAPH04] also employ the swirl detection and spanning tree heuristics described by [PSS01]. In addition, their method prefers the insertion of edges connecting "extremal vertices" with a large average geodesic distance to other landmarks.

As described in Sec. 4.1, greedy orderings are convenient to quickly derive global upper bounds on the total embedding length. We initialize our branch-and-bound algorithm by running all three greedy variants and choosing the best result as upper bound.

5.1. Layout Embedding in Shape Collections

Applications that establish correspondences within shape collections may require embedding a common layout into a range of models. This layout can carry semantic information, guide remeshing algorithms, or act as a parametrization base domain. Especially if the layout was created before all instances in the collection were known or when the collection contains outliers, insertion of new instances can be challenging and requires a robust embedding algorithm. We simulate this situation by generating one layout per class of the SHREC'07 dataset and embedding it into all models of that class.

Dataset From the SHREC'07 dataset [GBP07] with sparse landmark correspondences provided by [KLF11], we select all classes of genus 0 meshes with at least 3 landmark annotations (16 classes in total). For each class, we generate a triangular layout via constrained decimation of its first model: We incrementally perform halfedge collapses until only the landmark vertices (between 7 and 36 per class) remain. The resulting coarse mesh connectivity defines our prototypical template layout.

Experiment For each model, we run our algorithm as well as the three greedy methods based on [PSS01], [KSG03] and [SAPH04]. For visual clarity we apply the path smoothing operator of [PSS01] to all embeddings. Quantitative results (e.g. total embedding lengths) are reported prior to this post process.

Discussion In Fig. 11 we present a selection of resulting embeddings, while Fig. 10 shows quantitative results of the entire experiment: Per object class, we arrange all models within this class on the horizontal axis and plot the total embedding lengths achieved by the four methods on the vertical axis; i.e. each column represents a problem instance and each dot represents a solution by one of the

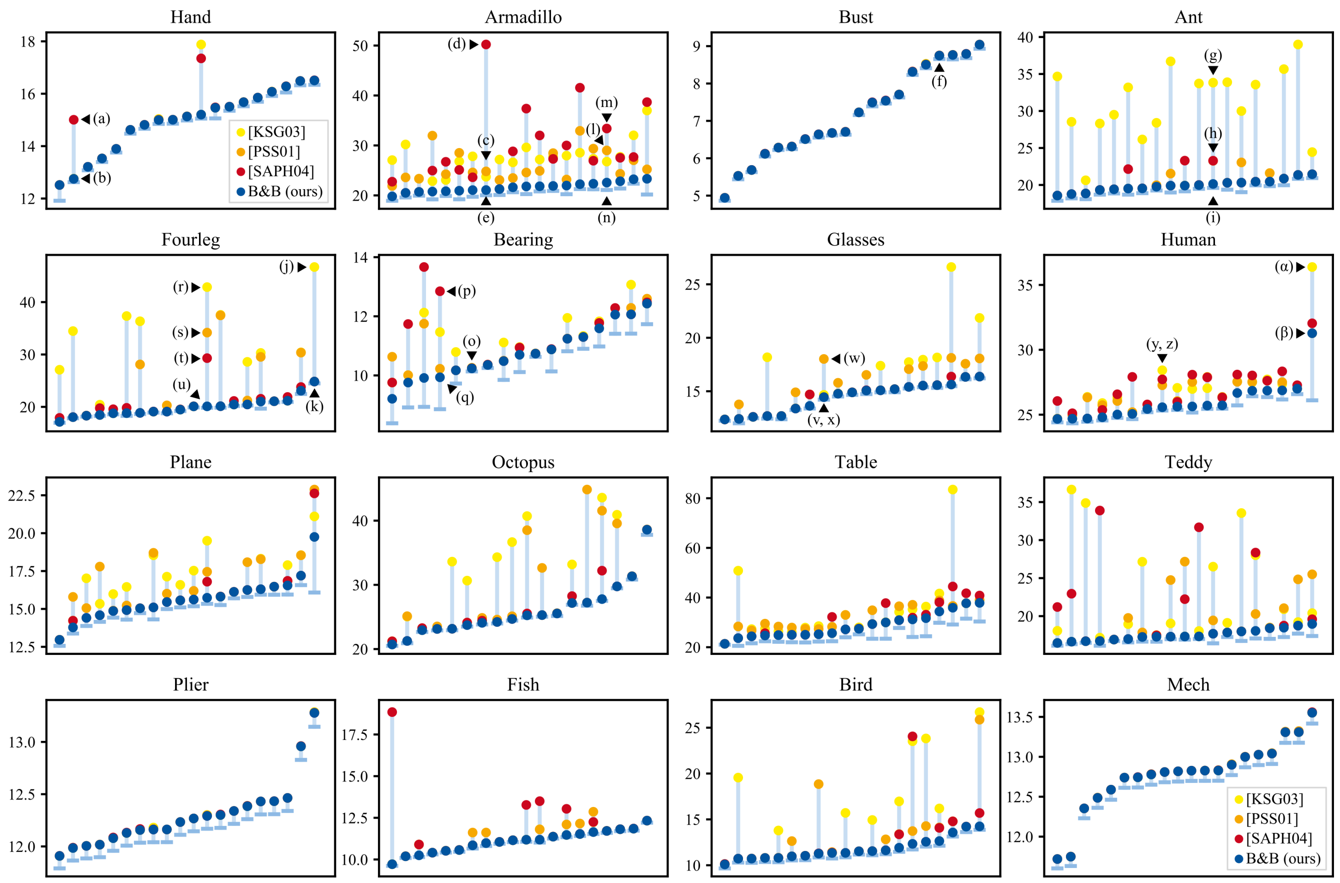


Figure 10: *Quantitative evaluation of our SHREC'07 experiment, in which we embed a common layout into the objects of each class. Corresponding qualitative results are reported in Fig. 11. Per class, we horizontally arrange all problem instances as columns and plot total embedding lengths on the vertical axis. Each dot represents a solution by one of the algorithms: either greedy—based on [KSG03] (yellow), [PSS01] (orange), [SAPH04] (red)—or by our branch-and-bound method (blue). Global lower bounds proven by our method are marked in light blue. For a few classes all methods perform equally well, but in most instances our branch-and-bound optimization achieves considerable improvements.*

algorithms. For each problem instance we mark the global lower bound $c_\perp$, computed by our method, with a horizontal bar (light blue). This means our solver proved that no embedding sequence shorter than this bound can exist.

As we employ all three greedy heuristics to find an initial upper bound, the embeddings computed by our method (blue dots in Fig. 10) are always shorter or equal to those of the greedy methods.

On favorable examples (e.g. Fig. 11 (f)), all four methods find the same embedding. In those cases (including the entire classes Bust, Plier and Mech, cf. Fig. 10), our algorithm quickly proves that a greedy embedding sequence is indeed optimal up to the prescribed threshold ($\alpha = 1\%$ in all examples).

In Fig. 10 we observe that each greedy method, while performing well in some instances, fails to generate shortest embeddings in a significant number of cases. The results shown in Fig. 11 demonstrate that these cases coincide with topologically unexpected embeddings containing excessive swirls.

While in some cases at least one greedy method yields the desired result (e.g. Fig. 11 (b), (f), (k)), there are plenty of cases in which all three fail (e.g. Fig. 11 (c-e), (g-i) (l-n), (r-u)). Therefore, an algorithm running all three heuristics and choosing the best result does not provide a satisfactory solution. In contrast, we did not find unnecessarily long paths or swirls in any of the embeddings produced by our branch-and-bound method.

In many cases (35% of the dataset, e.g. Fig. 10 (b), (f), (o), (z)) our algorithm confirms optimality of the solution (up to the threshold of 1%) within the given time frame (5 minutes in all examples). When the algorithm exhausts this time budget, small gaps are usually reported. For 83% of the dataset, optimality has been proven up to 5%.

Only in a few particularly challenging cases our algorithm was stopped with a large remaining gap (e.g. 10% in (q), 17% in (β)). Fig. 11 (q) poses a challenge as two landmark positions are swapped in the original dataset. Example (β) is difficult because it is the only model in the Human class where the arms are merged with upper body and legs (see zoom-in). While the task of embedding the given layout into this particular model is semantically questionable, our method still handles this outlier gracefully. Even though optimality is only proven up to 17%, we argue that the so-

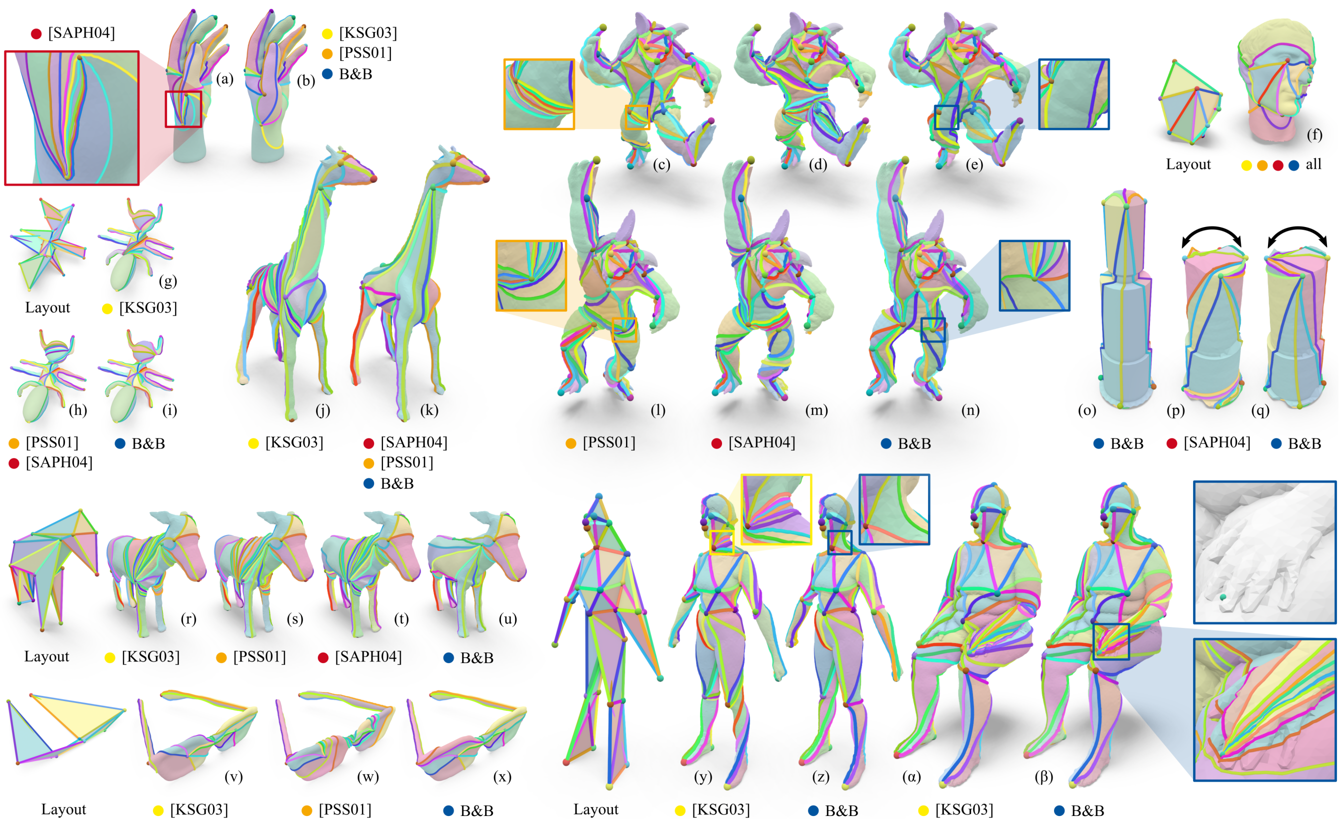


Figure 11: *Selected results of our evaluation in Fig. 10: Automatically generated layouts are embedded into models of the SHREC'07 dataset. We compare greedy methods based on the heuristics in [PSS01], [KSG03], and [SAPH04] against our branch-and-bound method. While in some cases (e.g. (f)) all methods achieve the desired result, and in some instances at least one greedy method succeeds ((b), (k)), there are many inputs on which all greedy strategies fail ((c-e), (g-i) (l-n), (r-u)). If embedding algorithms fail, defects can be severe, with paths forced into unwanted homotopy classes.*

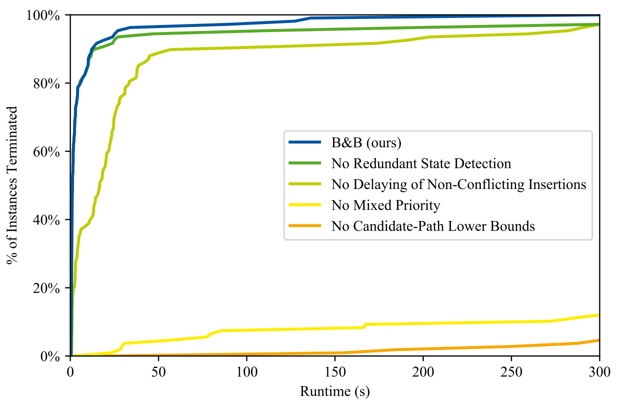


Figure 12: *Ablation study: We run our algorithm in different configurations (each disabling one of the features discussed in Sec. 4.3–4.5) on a subset of the SHREC'07 dataset. For each configuration, we report the percentage of instances which terminated within a given time.*

lution is indeed the desired one since we could not find any unnecessarily long paths or swirls upon visual inspection.

Leaving the difficult task of proving optimality aside, we observe that our algorithm produces good embeddings very quickly. In 60% of the dataset, the best solution of each problem instance was already found after 10 seconds. Only in 2% of cases the final result was found after more than 3 minutes. All timings were measured using a single-threaded implementation on a desktop computer.

Ablation Study We evaluate the performance impact of the branch-and-bound customizations discussed in Sec. 4. Based on the previous experiment, we select all instances of the SHREC'07 dataset where the search was completed within 5 minutes and rerun our algorithm in different configurations, each disabling one of its features. To rule out the influence of different heuristics, we do not initialize our algorithm with a greedy solution in this experiment. Results are reported in Fig. 12: The plot shows the relative number of instances that have terminated (i.e. found and proved an optimal solution up to 1%) within a given time. Disabling the detection of redundant states via hashing (dark green, Sec. 4.4.1) causes some runs to exceed the 5 minute time limit but has nearly no effect on simple inputs, where runtimes are close to our fully-featured al-

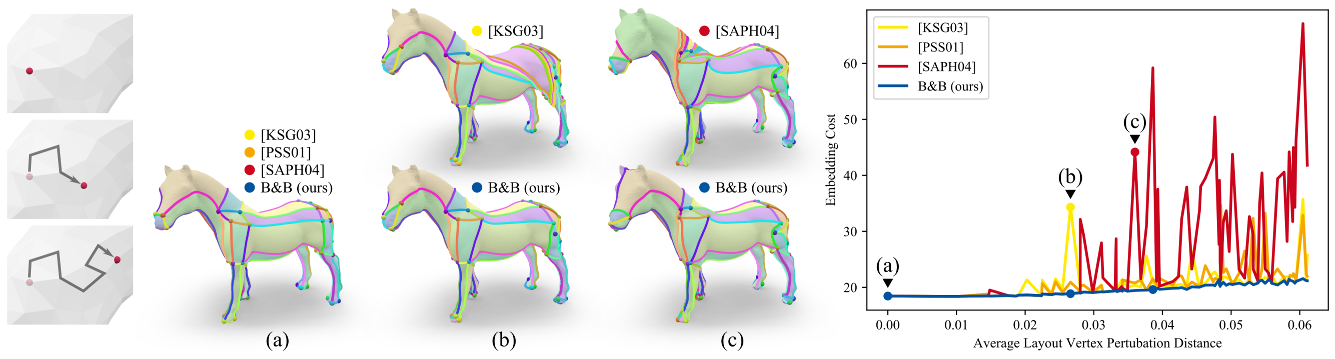


Figure 13: *Starting from their initial position on the target surface, we move landmarks along random trajectories and compute embeddings (a, b, c). Already for quite small perturbations (b, c), greedy methods (e.g. [KSG03, SAPH04]) can produce unfavorable changes in homotopy class, indicated by a discontinuous increase in embedding cost. In comparison, embeddings generated by our method (B&B) remain in the most adequate homotopy class (identical for (a, b, c)), and embedding costs increase only gradually.*

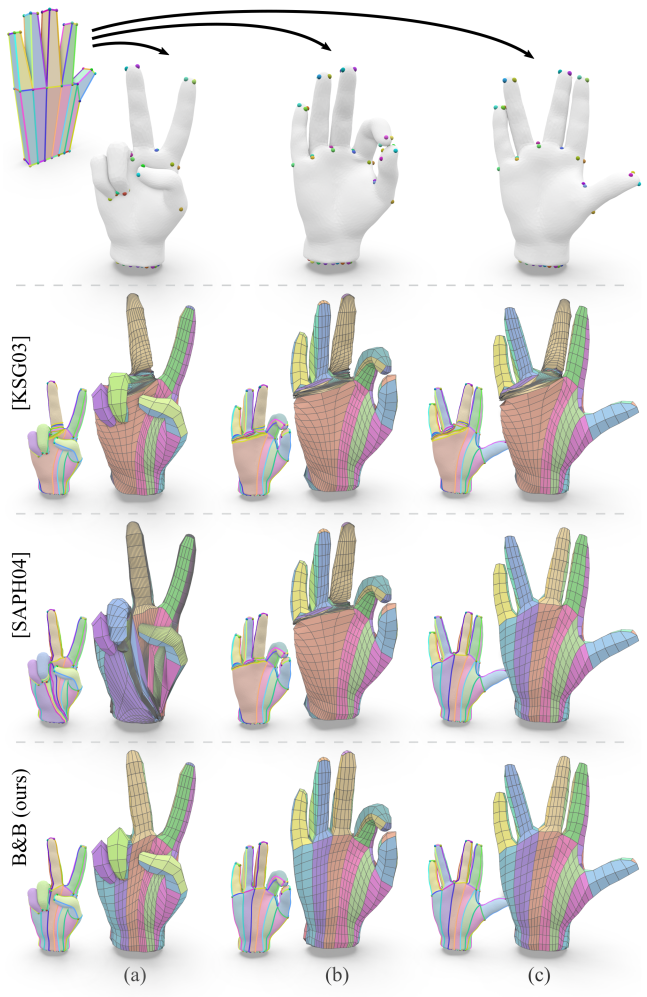


Figure 14: *We embed the same layout template into multiple target meshes (top row) to compute quad meshes (a, b, c) with prescribed base complex. Greedy methods, susceptible to excessive swirls, may produce embeddings of undesired homotopy, resulting in distorted quad meshes. Our branch-and-bound algorithm (bottom row) computes swirl-free embeddings in all examples, allowing us to extract clean quad meshes with the desired base complex.*

gorithm (blue). The impact of our proactive pruning by delaying non-conflicting insertions (Sec. 4.4.2) is more pronounced: When disabled (light green), performance deteriorates across almost all examples. With a traditional best-first search priority (yellow, instead of our mixed priority, Sec. 4.5), or a simplistic computation of lower bounds (orange, ignoring the cost of candidate paths, Sec. 4.3), only very few runs terminate within the time limit.

5.2. Robustness with Respect to Landmark Positions

In an additional experiment (Fig. 13) we compare the resilience of all four algorithms with respect to a perturbation of landmark positions on the target surface. Initially, we embed the layout using a favorable set of landmark positions. We then move each landmark along a random walk on the surface and recompute embeddings at different travel distances. We find that greedy methods are quite sensitive towards such changes in landmark positions. Already for mild perturbations, all three greedy methods introduce swirls leading to high embedding costs. Further, we observe that for an (approximately) continuous motion of target vertices, the length of greedy embeddings changes discontinuously, as path homotopy classes are often switched accidentally. In contrast, the lengths of our branch-and-bound embeddings increases gradually and all paths remain in their initial homotopy class for the duration of this experiment.

5.3. Quad Meshing with Prescribed Base Complex

Generating meshes with a high-quality base complex is a challenging task in parametrization-based quad meshing. The problem becomes even more difficult when the same resulting base complex is required across multiple target shapes. In many methods (e.g. [BZK09, BCE*13, RRP15]), the base complex arises as a result of the remeshing process on a single shape. An alternative approach is to reverse this dependency and explicitly model the base complex a priori; either manually [TPSHSH13, CK14a] or automatically [CBK12, ULP*15]. Several quad meshing methods (e.g. [BCE*13, CBK15, ESCK16]) offer control over the resulting base complex by constraining pairs of prescribed vertices

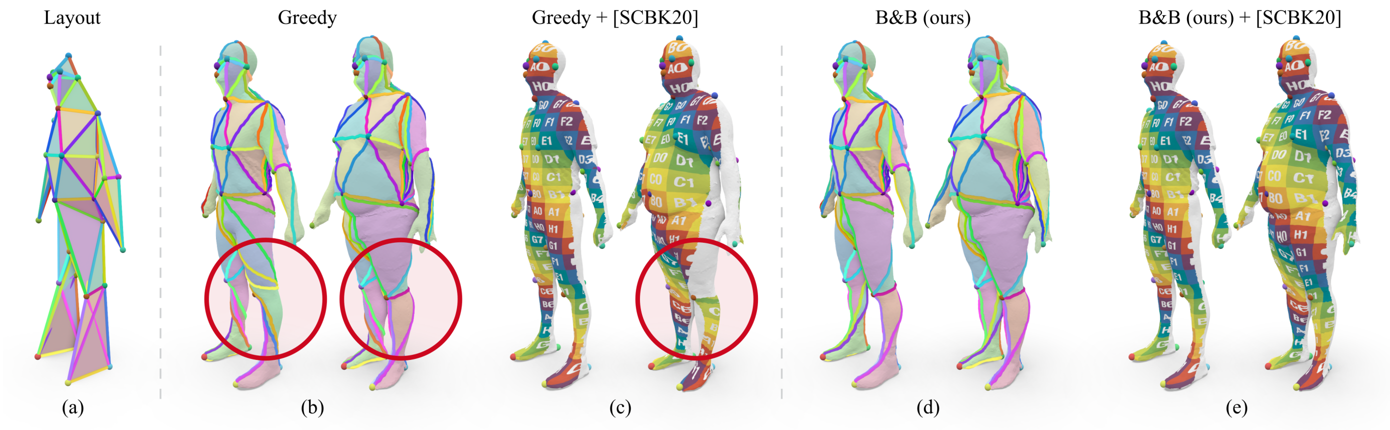


Figure 15: *We compute inter-surface maps by embedding an automatically generated layout (a) into two target models. Greedy embddings (here via [KSG03]) are likely to include paths in different homotopy classes across target models (see e.g. leg in (b)). Subsequent continuous map optimization (here using [SCBK20]) cannot leave the initial map homotopy class implied by the embeddings. We visualize the optimized map (c) by projecting a texture to the front-facing part of the first model and transferring it to the second model. Note how the map is twisted around the left leg, pulling parts of the untextured back to the front of the leg. Our method successfully generates shortest embeddings (d) which lead to the desired map homotopy class and allow continuous optimization to reach the expected result (e).*

to the same iso-parameter line. However, formulating such constraints requires knowledge of the desired path homotopy, which is usually not available when the target base complex was designed on a different model. Obtaining this required homotopy information amounts to the exact same problem as embedding a prescribed layout (base complex) into a target surface.

We illustrate the general approach of quad meshing with a prescribed base complex at the example of a simplistic quad meshing pipeline: We (1) embed a given quad layout into multiple target surfaces, (2) apply the path smoothing operator from [PSS01], (3) choose a number of subdivisions per dual loop, (4) Tutte-embed each patch to a planar rectangle, and (5) extract a quad mesh via regular re-sampling. In Fig. 14 we run this pipeline on three hand models and compare results using our embedding method to typical failure cases of greedy embeddings. While all resulting quad meshes share the same base complex, some meshes based on greedy embeddings exhibit extreme distortion due to paths and patches in undesirable homotopy classes. Such cases demonstrate that robustness of an embedding algorithm is essential when used as a step in a geometry processing pipeline. Our branch-and-bound algorithm produced the expected embedding homotopy in all our experiments.

5.4. Inter-Surface Map Initialization

Bijective maps between surfaces are often initialized in a patch-wise manner via compatible layout embeddings on two models [SAPH04, KS04, SCBK20]. The homotopy classes of paths on both surfaces together imply the homotopy class of the inter-surface map. If landmark correspondences are used as hard point-to-point constraints, this map homotopy is with respect to the surfaces punctured at the landmarks [APL15]. Continuous optimization algorithms reducing map distortion, by their very nature, cannot switch between map homotopy classes. Therefore, it is crucial to compute an initial map that is already in the desired homotopy class.

To demonstrate this importance, we pick a pair of models from the experiment in Sec. 5.1 and initialize inter-surface maps; once via greedy embeddings and once via our branch-and-bound embeddings (see Fig. 15). We then optimize both maps using [SCBK20] and visualize the results via texture transfer. The greedy pair (b) shows paths in different homotopy classes on the left leg: Some paths (e.g. yellow) are twisted around the leg of the first model, but not around the leg of the second model. As a result of this discrepancy, the optimized map (c) exhibits a twist, where regions from the back of the leg (untextured) are mapped to the front of the leg. The optimization [SCBK20] is inevitably trapped in this map homotopy class and cannot resolve the situation. In contrast, the embeddings produced by our optimal (up to 1%) insertion sequences solely contain paths in the expected homotopy classes (d) and lead to the desired bijective map (e).

6. Limitations and Future Work

Our method is limited to simply-connected (i.e., genus 0 or disk-topology, Fig. 16) domains. On other input topologies, our algo-

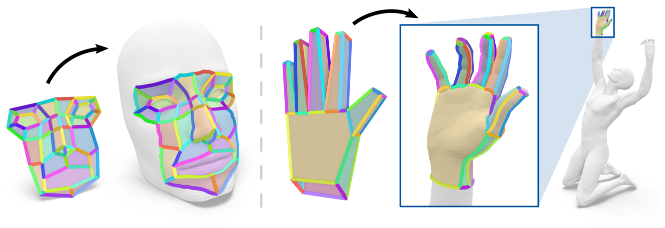


Figure 16: *Our algorithm can embed disk-topology layouts (face, hand) into target surfaces of disk topology (left) or genus 0 (right).*

rithm cannot guarantee to find a *cellular* embedding where all patches are disk-homeomorphic. A potential remedy is a strategy that constructs separating paths for every edge insertion which closes a layout cycle [LPVV01, SAPH04]. Incorporating these ideas into a branch-and-bound framework, however, poses additional challenges, e.g. exhaustively enumerating all relevant ways to route paths around handles.

One source of redundancy that remains undetected by our proactive pruning strategy (Sec. 4.4.2) arises when disjoint subsets of mutually conflicting edges can be resolved independently. In this case, insertions in different conflict components can be interleaved, leading to a number of sequences which can only later be identified as redundant (Sec. 4.4.1). A possible solution would be to solve component sub-problems separately. However, this is only viable if their individual results are guaranteed to remain non-conflicting, which is difficult to predict.

It would be interesting to explore objectives beyond total embedded path length. While different path metrics (e.g. favoring smoothness or feature alignment) can be readily integrated in our framework, a more challenging extension is the formulation of patch-based quality measures (e.g. mapping distortion), requiring new techniques for computing lower bounds.

A different direction is the application of our branch-and-bound optimization to more loosely constrained tasks such as shortest cut graph generation or compatible triangulation (without a prescribed layout). Here, one has to deal with a much higher branching degree, as decisions concerning the layout connectivity and its embedding need to be considered simultaneously. The even more challenging task of automatically embedding layouts *without* prescribed vertex locations is another interesting unsolved problem.

Acknowledgements

Open access funding enabled and organized by Projekt DEAL. [Correction added on 08 November 2021, after first online publication: Projekt Deal funding statement has been added.]

References

[APL14] Aigerman N., Poranne R., Lipman Y.: Lifted bijections for low distortion surface mappings. *ACM Transactions on Graphics 33*, 4 (2014). 2, 3

[APL15] Aigerman N., Poranne R., Lipman Y.: Seamless surface mappings. *ACM Transactions on Graphics 34*, 4 (2015). 2, 3, 12

[BCE*13] Bommes D., Campen M., Ebke H.-C., Alliez P., Kobbelt L.: Integer-grid maps for reliable quad meshing. *ACM Transactions on Graphics 32*, 4 (2013). 3, 11

[BF98] Bastert O., Fekete S. P.: *Geometric Wire Routing*. Tech. rep., University of Cologne, 1998. 3

[BWK05] Bischoff S., Weyand T., Kobbelt L.: Snakes on triangle meshes. In *Bildverarbeitung für die Medizin*. Springer, 2005. 3

[BZK09] Bommes D., Zimmer H., Kobbelt L.: Mixed-integer quadrangulation. *ACM Transactions on Graphics 28*, 3 (2009). 11

[Cam17] Campen M.: Partitioning surfaces into quadrilateral patches: a survey. *Computer Graphics Forum 36*, 8 (2017). 3

[CBK12] Campen M., Bommes D., Kobbelt L.: Dual loops meshing: Quality quad layouts on manifolds. *ACM Transactions on Graphics 31*, 4 (2012). 3, 11

[CBK15] Campen M., Bommes D., Kobbelt L.: Quantized global parametrization. *ACM Transactions on Graphics 34*, 6 (2015). 11

[CdV10] Colin de Verdière É.: Shortest cut graph of a surface with prescribed vertex set. In *European Symposium on Algorithms* (2010), Springer. 3

[CHKL13] Chan T. M., Hoffmann H.-F., Kiazyk S., Lubiw A.: Minimum length embedding of planar graphs at fixed vertex locations. In *International Symposium on Graph Drawing* (2013), Springer. 3

[CK14a] Campen M., Kobbelt L.: Dual strip weaving: Interactive design of quad layouts using elastica strips. *ACM Transactions on Graphics 33*, 6 (2014). 3, 11

[CK14b] Campen M., Kobbelt L.: Quad layout embedding via aligned parameterization. *Computer Graphics Forum 33*, 8 (2014). 3

[CPL09] CPLEX: User's manual for CPLEX. *International Business Machines Corporation 46*, 53 (2009). 3

[DBG*06] Dong S., Bremer P.-T., Garland M., Pascucci V., Hart J. C.: Spectral surface quadrangulation. In *Proceedings of SIGGRAPH '06* (2006). 3

[EH96] Eck M., Hoppe H.: Automatic reconstruction of B-spline surfaces of arbitrary topological type. In *Proceedings of SIGGRAPH '96* (1996). 3

[EHP04] Erickson J., Har-Peled S.: Optimally cutting a surface into a disk. *Discrete & Computational Geometry 31*, 1 (2004). 3

[ESCK16] Ebke H.-C., Schmidt P., Campen M., Kobbelt L.: Interactively controlled quad remeshing of high resolution 3d models. *ACM Transactions on Graphics 35*, 6 (2016). 11

[EW05] Erickson J., Whittlesey K.: Greedy optimal homotopy and homology generators. In *SODA* (2005), vol. 5. 3

[FM11] Farb B., Margalit D.: *A Primer on Mapping Class Groups*. Princeton University Press, 2011. 3

[GAB*20] Gamrath G., Anderson D., Bestuzheva K., Chen W.-K., et. al: *The SCIP Optimization Suite 7.0*. Tech. rep., Optimization Online, 2020. 3

[GBP07] Giorgi D., Biasotti S., Paraboschi L.: SHape REtrieval Contest 2007: Watertight models track, 2007. 8

[GO20] Gurobi Optimization L.: Gurobi optimizer reference manual, 2020. 3

[HS94] Hershberger J., Snoeyink J.: Computing minimum length paths of a given homotopy class. *Computational Geometry 4*, 2 (1994). 3

[KLF11] Kim V. G., Lipman Y., Funkhouser T.: Blended intrinsic maps. *ACM Transactions on Graphics 30*, 4 (2011). 8

[KLS03] Khodakovsky A., Litke N., Schröder P.: Globally smooth parameterizations with low distortion. *ACM Transactions on Graphics 22*, 3 (2003). 3

[KS04] Kraevoy V., Sheffer A.: Cross-parameterization and compatible remeshing of 3D models. *ACM Transactions on Graphics 23*, 3 (2004). 2, 3, 8, 12

[KSG03] Kraevoy V., Sheffer A., Gotsman C.: Matchmaker: Constructing constrained texture maps. *ACM Transactions on Graphics 22*, 3 (2003). 2, 3, 8, 9, 10, 11, 12

[Liv20] Livesu M.: Scalable mesh refinement for canonical polygonal schemas of extremely high genus shapes. *IEEE Transactions on Visualization and Computer Graphics (TVCG)* (2020). 3

[LL02] Lee Y., Lee S.: Geometric snakes for triangular meshes. *Computer Graphics Forum 21*, 3 (2002). 3

[LMM*95] Liebling T. M., Margot F., Müller D., Prodon A., Stauffer L.: Disjoint paths in the plane. *ORSA Journal on Computing 7*, 1 (1995). 3

[LPVV01] LAZARUS F., POCCHIOLA M., VEGTER G., VERROUST A.: Computing a canonical polygonal schema of an orientable triangulated surface. In *Proceedings of the Seventeenth Annual Symposium on Computational Geometry* (2001). 13

[LSS*98] LEE A. W. F., SWELDENS W., SCHRÖDER P., COWSAR L., DOBKIN D.: MAPS: Multiresolution adaptive parameterization of surfaces. In *Proceedings of SIGGRAPH '98* (1998). 3

[MJSS16] MORRISON D. R., JACOBSON S. H., SAUPPE J. J., SEWELL E. C.: Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization 19* (2016). 3

[PSS01] PRAUN E., SWELDENS W., SCHRÖDER P.: Consistent mesh parameterizations. In *Proceedings of SIGGRAPH '01* (2001). 2, 3, 4, 8, 9, 10, 12

[RRP15] RAZAFINDRAZAKA F. H., REITEBUCH U., POLTHIER K.: Perfect matching quad layouts for manifold meshes. *Computer Graphics Forum 34*, 5 (2015). 3, 11

[SAPH04] SCHREINER J., ASIRVATHAM A., PRAUN E., HOPPE H.: Inter-surface mapping. In *ACM SIGGRAPH 2004 Papers* (2004). 2, 3, 4, 8, 9, 10, 11, 12, 13

[SBLS18] SORGENTE T., BIASOTTI S., LIVESU M., SPAGNUOLO M.: Topology-driven shape chartification. *Computer Aided Geometric Design 65* (2018). 3

[SC20] SHARP N., CRANE K.: You can find geodesic paths in triangle meshes by just flipping edges. *ACM Transactions on Graphics 39*, 6 (2020). 3

[SCBK20] SCHMIDT P., CAMPEN M., BORN J., KOBBELT L.: Inter-surface maps via constant-curvature metrics. *ACM Transactions on Graphics 39*, 4 (2020). 2, 3, 12

[TPSHSH13] TAKAYAMA K., PANOZZO D., SORKINE-HORNUNG A., SORKINE-HORNUNG O.: Sketch-based generation and editing of quad meshes. *ACM Transactions on Graphics 32*, 4 (2013). 3, 11

[ULP*15] USAI F., LIVESU M., PUPPO E., TARINI M., SCATENI R.: Extraction of the quad layout of a triangle mesh guided by its curve skeleton. *ACM Transactions on Graphics 35*, 1 (2015). 3, 11

[YSC20] YU C., SCHUMACHER H., CRANE K.: Repulsive curves, 2020. arXiv:2006.07859. 3

Appendix A: Proof: Delaying Non-Conflicting Insertions

Consider a state s and a pair of edges $c \in C(s)$ and $n \in \overline{C}(s)$. Since n is non-conflicting, its insertion will neither change the candidate path of any other edge when inserted, nor will its candidate path be changed by the insertion of any other edge into s, i.e.

$$p(M(s),c) = p(M(sn),c),$$
$$p(M(s),n) = p(M(sc),n),$$

which implies $M(snc) = M(scn)$, and thus

$$M(sncs') = M(scns') \tag{1}$$

for any remaining sequence s'.

Our conflict-avoiding pruning only visits states $v \in V$ with insertion sequences of the form $v = s_c s_n$ where

- s_c is a *conflicting sequence*, i.e. for each prefix $s'_c e \sqsubseteq s_c$, it is $e \in C(s'_c)$,
- s_n is a *non-conflicting sequence*, i.e. for each prefix $s_c s'_n e \sqsubseteq s_c s_n$, it is $e \in \overline{C}(s_c s'_n)$.

We show that for every unvisited state $s \notin V$, there is an embedding sequence $v \in V$ such that $M(s) = M(v)$. We can construct v from s by iteratively pushing non-conflicting insertions towards the end:

Suppose $s \notin V$. Then, there is a decomposition $s = s_c s_n n c s'$ such that s_c is a (possibly empty) conflicting sequence, s_n is a (possibly empty) non-conflicting sequence, $n \in \overline{C}(s_c s_n)$, and $c \in C(s_c s_n)$. By Eq. (1), we have $M(s_c s_n n c s') = M(s_c s_n c n s')$. Repeating this operation yields $M(s_c s_n n c s') = M(s_c c s_n n s')$. As $c \in C(s_c s_n)$, so is $c \in C(s_c)$, hence $s_c c$ is a conflicting sequence. If $s_n n s'$ is a non-conflicting sequence, then $s_c c s_n n s' \in V$ (*q.e.d.*). Otherwise, we repeat the same argument for the remaining part until s' becomes empty.