6.838: Shape Analysis, 2021
Instructor: Justin Solomon
TAs: David Palmer and Paul Zhang

Towards an Algorithm for Reeb Graph Construction on Constructive Solid Geometry

Logan Weber[1]

[1]MIT Department of Electrical Engineering and Computer Science

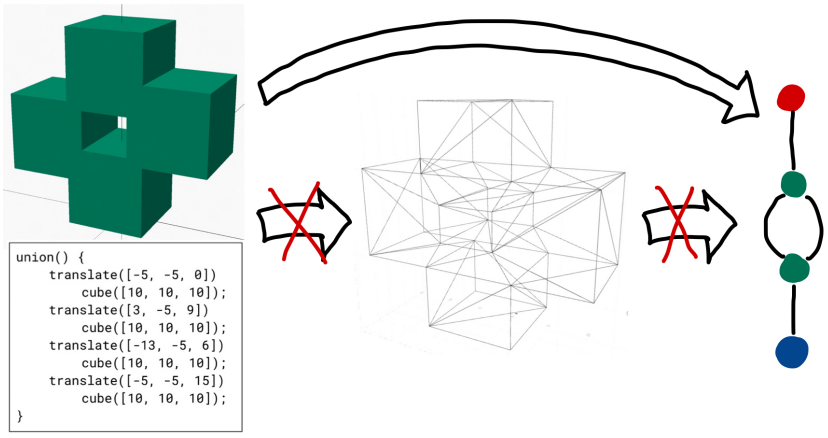


Figure 1: Rather than converting constructive solid geometry (CSG) into a triangle mesh to generate a Reeb graph, our algorithm computes Reeb graphs directly from a subset of CSG.

Abstract

*Ever since Shinagawa, Kunii, and Kergosien gave the first algorithm for computing Reeb graphs [SKK91], they have become an important technique in computational topology. Their work initially drew interest from the computer graphics community and has led to new algorithms for constructing Reeb graphs, new theory for studying them, and new applications in which to utilize them. In all published work involving the construction of Reeb graphs on **CAD models**, the models are first converted into triangulated surfaces, then standard construction algorithms are applied to the surface. However, constructing Reeb graphs **directly** from CAD representations like constructive solid geometry (CSG) could give new insights into the properties of Reeb graphs and give rise to a new class of algorithms for generating them. As a first step in realizing this goal, we present an algorithm for constructing Reeb graphs of height functions **directly** from a subset of CSG.*

1. Introduction

Reeb graphs, originally proposed by Georges Reeb in 1946 [Ree46], capture the level set topology of a real-valued function on a manifold. They were originally a tool of **Morse theory**, which aims to study the topology of a manifold by studying differentiable functions on it.

Reeb graphs didn't see much light in the realm of computer science until the work of Shinagawa, Kunii, and Kergosien in 1991, wherein they presented the first algorithm for automatically constructing Reeb graphs, with applications in medical imaging [SKK91]. Reeb graphs garnered further interest when Hilaga et al. demonstrated their efficacy in computing shape similarity [HSKK01]. Ever since, Reeb graphs have found applications throughout shape analysis [BGSF08], and more recently, topological data analysis [Mun17].

Thereafter, in the flurry of activity around Reeb graphs, Bespalov, Regli, and Shokoufandeh applied Reeb graphs to the shape retrieval task for *CAD models* [BRS03]. In this work, they applied

the work of Hilaga et al. with minimal modification and showed the promise of Reeb graphs in the CAD domain as well.

Despite having access to domain-specific representations (e.g., boundary representations and constructive solid geometry (CSG)), all prior work in the CAD realm constructs Reeb graphs from triangle meshes by first converting from a CAD representation. The reason for doing so is economical, as a triangle mesh provides a common format to compile the various CAD formats to. However, computing directly from CSG may be of interest for several reasons: **(1)** It provides a different view of Reeb graphs that may be of theoretical interest. In particular, it may be valuable to understand how topology changes under the set-theoretic operations present in CSG. **(2)** It gives rise to a new class of algorithms for Reeb graph construction with different runtime complexities.

In this paper, we recapitulate a Reeb graph construction algorithm based on ascending paths on triangulated surfaces, then we propose a new algorithm for constructing Reeb graphs directly from a subset of CSG expressions (namely, the restriction to cube primitives and the union operation).

2. Background

If $f : M \to \mathbb{R}$ is a real-valued function on a compact manifold M, then we start to define the **Reeb graph** of (M, f) by first defining the equivalence relation $x \sim y$, which holds when $f(x) = f(y)$ and x and y belong to the same connected component at $f^{-1}(f(x))$. Then the Reeb graph is the quotient space $M/\sim$.

As Morse theory is concerned with *differentiable* functions, it is often desired that f is differentiable, and additionally, that it has no degenerate critical points. A function f that satisfies these properties is called a **Morse function**.

Despite the desire to work with Morse functions, functions defined on arbitrary surfaces are likely to have degenerate critical points. However, even though Reeb graphs are a tool of Morse theory, we don't need Morse functions in order to compute the evolution of the level set topology. This idea is explored in [BFS00], where Biasotti, Falcidieno, and Spagnuolo contribute the insight that degenerate critical points form critical areas, and all points within a simply connected critical area are **Reeb-equivalent**, meaning they can be collapsed into a single node.

A popular choice of function for Reeb graphs on surfaces is the **height function**, which is an *extrinsic* function (i.e., dependent upon the particular embedding of the surface within a manifold). The height function $h : M \to \mathbb{R}$ of a manifold $M \subseteq \mathbb{R}^3$ is defined as $h(\boldsymbol{p}) = h((x,y,z)) = y$. For the sake of simplicity, we restrict our attention to the height function in this paper, though there is work that uses functions defined in terms of *intrinsic* properties (e.g., average geodesic distance to all other points on the mesh), which give Reeb graphs that are invariant to rigid motion and scaling [HSKK01]. The Reeb graph of the height function on a torus is shown in Figure 2.

3. Related Work

Reeb graphs are of a theoretical nature and date back to 1946 when Georges Reeb proposed them as a tool in Morse theory [Ree46],

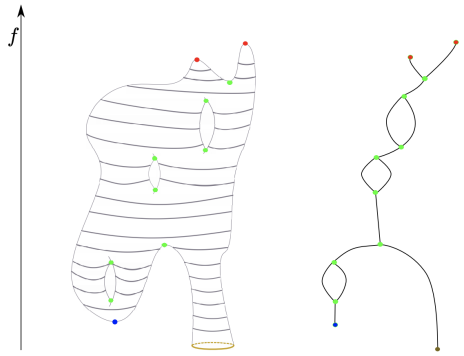


Figure 2: Example of a 2D surface with boundary (left) and the Reeb graph of the height function f on this surface (right). Image source: [HR20]

which aims to study the topology of manifolds by defining differentiable functions on them and examining their behavior.

It should be noted that **contour trees** are sometimes mentioned in the literature, but they are a less general object. In essence, contour trees are Reeb graphs on simply connected domains, and hence have no cycles. When they show up in the literature, they are often given less theoretical motivation and are geared towards applications.

Shape Reconstruction. The first algorithm for Reeb graph construction was by Shinagawa and Kunii in 1991 [SKK91]. Their application was to medical imaging, and the setting differs significantly from later applications. Their problem setup is that they wish to construct the Reeb graph of a **height function** defined on a shape, given only cross-sectional contours (approximated as polygons) of the shape. Once they have the Reeb graph, their goal is 3D shape reconstruction, and their driving example is reconstruction of a human cochlea. The algorithm runs in $\mathcal{O}(n^2)$ time, where n is the number of total points in the contours.

Shape Similarity. In a seminal 2001 paper, Hilaga et al. proposed *shape similarity* as a viable application for Reeb graphs [HSKK01]. The input is different from Shinagawa and Kunii's, since in the work of Hilaga et al., they assume the input is a triangle mesh. Their justification for triangle meshes is that they can be converted to and from other representations. In particular, they developed a variation called a **multiresolutional Reeb graph** (MRG) which captures topological structure at multiple levels of detail. They also proposed an *intrinsic* alternative to the *extrinsic* height function. In particular, at each point, they define $f : S \to \mathbb{R}$ as

$$f(\boldsymbol{v}) = \frac{\mu(\boldsymbol{v}) - \min_{\boldsymbol{p}\in S}\mu(\boldsymbol{p})}{\max_{\boldsymbol{p}\in S}\mu(\boldsymbol{p})}$$

where $\mu(\boldsymbol{v}) = \int_{\boldsymbol{p}\in S} g(\boldsymbol{v},\boldsymbol{p})dS$ is the sum of geodesic distances from $\boldsymbol{v}$ to all other points $\boldsymbol{p}$. That is, they use the sum of geodesic distances so the Reeb graph is invariant to rigid motion, and they normalize so it's invariant to scale. They approximate the geodesic distance by running Dijkstra's algorithm on a mesh with shortcut edges (to improve the approximation). The similarity between two

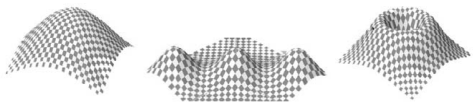


Figure 3: Example of a height function with a single isolated maximum (a), a height function with a critical point of multiplicity 3 (b), and a height function with a connected component of maxima (c). Image source: [BFS00]

shapes is then computed using a custom weighted graph matching algorithm. They are able to leverage the multiresolutional structure of the MRG to iteratively build solutions, starting from the coarsest resolution (a single node) where the solution is trivial, then simultaneously refining the resolution and the mapping. The runtime of their construction algorithm is $\mathcal{O}(n+c)$, where n is the number of vertices of the mesh and c is the number of vertices inserted while generating contours at each resolution.

Since Hilaga et al.'s paper, there has been a proliferation of work on parallel construction algorithms [PCM03, HR20], online construction algorithms [PSBM07], randomized construction algorithms [HWW10], miscellaneous construction algorithms [DN09, DN13], the study of distances between Reeb graphs [BB13, BGW14, BMW15, FL16, SMP16, BFL16, BLM20], and the application of Reeb graphs to topological data analysis [Mun17, TFL*18].

Extended Reeb Graphs. Morse theory cares about Morse functions (functions with no degenerate critical points), but there's a problem with using Reeb graphs on discrete surfaces, because discrete surfaces might have many degenerate critical points. Early techniques proposed "hacks" to fix this problem. For example, when considering the Reeb graph of a height function, you can often rotate the shape slightly to remove degenerate critical points. The authors of [BFS00] note that this kind of fix may patch up the theoretical problem, but it introduces nodes in the Reeb graph that do not correspond to shape features. To fix this problem, they propose **extended Reeb graphs** (ERGs), which handle degenerate critical points by generalizing to critical *areas*, and they propose an algorithm for constructing ERGs. Figure 3(c) shows an example of a critical area.

Computer-Aided Design. Reeb graphs were first applied to CAD in 2003 by Bespalov, Regli, and Shokoufandeh [BRS03]. In this work, they directly followed the work of Hilaga et al., and their goal was simply to show the efficacy of Reeb graphs for shape retrieval on solid models and to present a dataset of solid models to test against.

In 2006, Biasotti et al. extended this technique for model retrieval and sub-part correspondence [BMSF06]. Their contribution is a method for partial shape matching that is able to recognize similar sub-parts of objects by making use of Spherical Harmonics. This work differs in that the method emphasizes a sub-part similarity measure *in addition to* global similarity measures given by prior work.

Present Work. All work the author could find that applied Reeb graphs to CAD assumes the input is a polygonal mesh, rather than a solid model format (e.g., a boundary representation, constructive solid geometry, or a mixture thereof). Thus, the present work differentiates itself from prior work in that we compute Reeb graphs of height functions **directly** from constructive solid geometry expressions.

Interestingly, this algorithm can be thought of as most similar to the orignal work of Shunigawa, Kunii, and Kergosien [SKK91], since in their work, they are constructing the Reeb graph directly from cross-sectional contours, rather than from a triangle mesh. An important difference is that, by having access to the CSG expression, we can be more precise in how we choose the contours to generate.

Further Reading. The author defers further discussion to an extensive survey paper by Biasotti et al. on Reeb graphs and their applications to shape analysis [BGSF08].

4. Technical Approach

In this section, we present an algorithm for constructing Reeb graphs from a subset of **constructive solid geometry**, wherein we allow only cubes as primitives and union as an operation.

We derived our implementation by analogy to the algorithm described in [HR20] for Reeb graph construction on **triangle meshes**, which we henceforth term the "ascending paths" method. Thus, to build intuition for the former, we describe the ascending paths algorithm in Section 4.1, then we present the algorithm for CSG in Section 4.2. Then, in Section 5, we compare the Reeb graphs generated by both techniques.

4.1. Reeb Graph Construction on Triangle Meshes

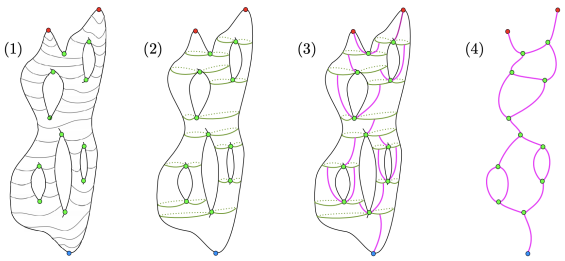


Figure 4: Summary of the ascending paths method. In step (1), the critical points of the input surface are calculated. Then, in step (2), the critical sets are computed. In step (3), ascending paths are computed from each saddle point and minimum and . When an ascending path contacts a critical set, the source of the ascending path is connected to the critical point corresponding to the critical set, giving the resulting Reeb graph (4). Image source: [HR20]

The ascending paths method involves three core computational components: critical points, **critical sets**, and ascending paths. We describe each separately before presenting the entire algorithm.

Critical Points. Formally, a point $\boldsymbol{p} \in S$ is a critical point of f if the differential $df_{\boldsymbol{p}}$ is zero. On triangle meshes, we can check if a point $\boldsymbol{p}$ is critical by analyzing the sign of $f(\boldsymbol{v}) - f(\boldsymbol{p})$ for each point $\boldsymbol{v}$ in the 1-ring of $\boldsymbol{p}$. The cases are shown in Figure 5.

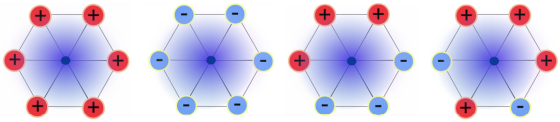


Figure 5: Types of sign configurations in the 1-ring of a vertex: (left) a minimum, (left middle) a maximum, (right middle) a regular vertex, and (right) a saddle point. Image source: [HR20]

Critical Sets. A critical set $C_{\boldsymbol{p}}$ is defined with respect to a critical point $\boldsymbol{p}$ and is an approximation of the level set of $f(\boldsymbol{p})$. On a triangle mesh, the true level set is given by a polyline curve over the mesh faces, however, *exactly* representing this curve is not necessary, as we can form an overapproximation of it as a set of vertices. Namely, for each edge the curve *would* travel through, we include the endpoint vertices, one of which will certainly have a *higher* f value, and one of which will certainly have a *lower* f value (see Figure 7). To ease the computation of ascending paths in the algorithm, we keep track of each stable component separately, rather than storing the critical set in aggregate (see Figure 6). We will see why later.

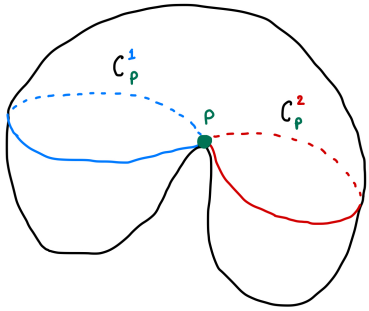


Figure 6: An illustration of critical set components $C_{\boldsymbol{p}}^1, C_{\boldsymbol{p}}^2$. The aggregate critical set $C_{\boldsymbol{p}}$ is given by $\bigcup_i C_{\boldsymbol{p}}^i = C_{\boldsymbol{p}}^1 \cup C_{\boldsymbol{p}}^2$.

Critical sets for minima and maxima consist of only the critical point themselves, and are thus trivial to compute. To compute critical sets for saddles, we start with the critical point $\boldsymbol{p}$, then within the 1-ring of $\boldsymbol{p}$, we find all pairs of adjacent points $\boldsymbol{u}$, $\boldsymbol{v}$ for which $\text{sign}(f(\boldsymbol{u}) - f(\boldsymbol{p})) \neq \text{sign}(f(\boldsymbol{v}) - f(\boldsymbol{p}))$ (i.e., one is above $\boldsymbol{p}$ and one is below $\boldsymbol{p}$). These pairs form the starts of the paths that the level set curve passes between.

We now restrict our attention to a **single** pair. For each pair, we choose the point $\boldsymbol{v}$ that is below $\boldsymbol{p}$ as the start of the path. At each step, we add $\boldsymbol{v}$ to the critical set and any neighbors that are adjacent to $\boldsymbol{v}$ and above $\boldsymbol{p}$. Then for the next step, we choose the neighbor of $\boldsymbol{v}$ that is *below* $\boldsymbol{p}$ and that is adjacent to a point that is *above* $\boldsymbol{p}$.

We perform this walk for each pair described earlier, and each walk gives a component $C_{\boldsymbol{p}}^i$ of $C_{\boldsymbol{p}}$. After all walks, we have computed the aggregate critical set $C_{\boldsymbol{p}}$.

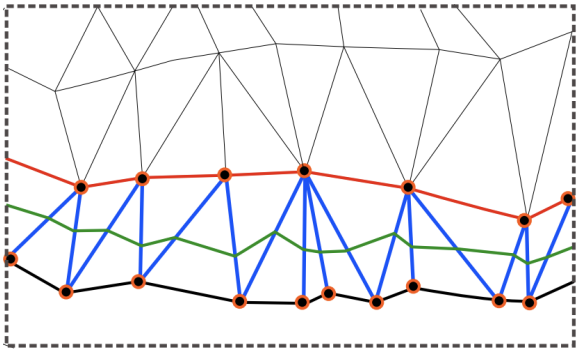


Figure 7: Close-up of a critical set (points circled orange) determined by the polyline curve (green), which represents the level set of the height function at a particular altitude. Image source: [HR20]

Ascending Paths. Ascending paths are paths starting at critical points that travel upwards with respect to the height function until they meet a critical set (they are guaranteed to meet a critical set eventually).

To compute ascending paths, we initialize the start vertex to a critical point, then we greedily choose the neighbor in our 1-ring with the highest value as the next node in our path. At each step, we check if the current node is contained in any point's critical set, and if it is, finish.

The Algorithm. A summary of the algorithm is given in Figure 4. To begin, we first compute the critical points of the mesh. Each critical point becomes a node in the resulting Reeb graph. For each critical point, we compute its critical set. We then mark each component of each critical set as unvisited.

Now, we compute ascending paths. Note that we cannot have ascending paths for maxima (there's no way to go up!), so we need only consider minima and saddle points. For a minimum, we compute a single ascending path and connect the minimum to whichever critical set it reaches. For a saddle, we send two ascending paths from $\boldsymbol{p}$, and for each unvisited $C_{\boldsymbol{v}}^i$ touched by a path, an edge is added from $\boldsymbol{p}$ to $\boldsymbol{v}$ and $C_{\boldsymbol{v}}^i$ is marked as visited. If we had not stored each component of critical sets separately, we would need to reason about which type of saddle (split or merge) we are sending paths from, in order to know *how many* to send. Storing and using the critical sets like so has the additional benefit that we can handle saddles of higher multiplicity with no further modifications.

4.2. Reeb Graph Construction on CSG

We now present an algorithm for constructing Reeb graphs from CSG expressions consisting of unions of cubes. The algorithm can be summarized as follows:

1. Identify the critical points of all primitives in the expression.
2. Iterate over critical points in order of height.
   a. Compute collisions of primitives with plane at $y-\varepsilon$ and $y+\varepsilon$.

b. Determine connectivity change from $y-\varepsilon$ to $y+\varepsilon$.
c. Add nodes/edges depending on type of connectivity change.

The full algorithm is given in Algorithm 1 and an example execution is given in Figure 8. In the rest of this section, we provide the concepts necessary to understand the algorithm, and as seen fit, we expand on particular steps of the algorithm.

Critical Areas. The algorithm first computes the critical areas of all primitives, which is as simple as gathering all primitives and collecting their critical areas. We can precompute the critical areas for each primitive, and since we only consider cubes, we know each primitive will only have two critical areas: the top and bottom faces. Once we have the critical areas, we sort them by their height, throw away duplicates, and begin sweeping over them. See Figure 8(b).

Critical Difference. A key difference from the ascending paths method is that, with our algorithm, not all of the primitive critical areas correspond to the critical areas of the overall shape. That is, the primitive critical areas provide an *overapproximation* of the true set of critical areas. In Figure 8, for example, we see steps (2), (3), (6), and (7) have no corresponding node in the resulting Reeb graph. A corollary of this fact is that we do not know a priori what nodes (or how many) we will have in our final Reeb graph, so we need to create nodes dynamically.

Collisions With Planes $\approx$ Critical Sets. At the critical point with y-coordinate y_i, we compute the collisions of *all* primitives with the plane $y = y_i + \varepsilon$ above the critical point and $y = y_i - \varepsilon$ below the critical point. These collisions give rise to 2D shapes on the plane, so we can use standard 2D collision detection algorithms to determine which shapes belong to the same level set component.

Representing Connected Components of Collisions. To represent the components, we use a union-find (UF) data structure, and while iterating over all pairs of shapes, if two shapes are determined to intersect, we union them together. These union-find structures are analogous to the critical sets of the ascending paths method, but a subtle difference is that we don't aim to store the level set topology of the critical point *itself* at $y = y_i$, but rather the level set topology above and below it. While critical sets **may** store contours that include vertices above and below, this is only due to the need to approximate the true polyline curve that represents the contour at $y = y_i$ (see Figure 7).

Frontiers $\approx$ Ascending Paths. The analogy to ascending paths here is the idea of a **frontier**. The frontier maps from each primitive to the vertex that spawned the path to it. The purpose is, when a topology change occurs, to know which source vertex to connect to the generated vertex in the Reeb graph. It is important to note that the frontier is simultaneously computing all ascending paths that would be active at the current y level as it scans upwards. Contrast this property to the computation of ascending paths, in which a single path is computed at a time.

We continue "ascending paths" from the previous iteration by comparing the previous frontier to the current frontier, and if a topology change occurs, creating a vertex and terminating the current path, or otherwise continuing the path upwards. To determine whether there was a topological change, we need to compare the components above and below the plane. We term the union-find structures that represent these respective components as the $(y+\varepsilon)$-UF and the $(y-\varepsilon)$-UF.

Comparing Above To Below. We first consider the $(y+\varepsilon)$-UF. For each component c^+ of the $(y+\varepsilon)$-UF, we find the components $\{c_i^-\}$ that its members stem from by performing a find query on the $(y-\varepsilon)$-UF. If there are no components that c^+ stems from, then c^+ must be a local min, in which case we create a new vertex v and map all members of c^+ to v in the frontier (see step (1) of Figure 8). If there is only **one** component c^- that c^+ stems from, then there is no topology change, and we map all members of c^+ in the current frontier to the vertex c^- maps to in the previous frontier—that is, we thread the mapping upwards (see steps (2), (3), (6), and (7) in Figure 8). Otherwise, if there are **multiple** components $\{c_i^-\}$ that c^+ stems from, then we are at a *join* point. Here, we insert a vertex into the graph, and for each component c^- in $\{c_i^-\}$, we add an edge from the vertex c^- maps to in the previous frontier to the new vertex. Then, in the current frontier, we map all members to the new vertex (see step (5) of Figure 8).

Comparing Below To Above. Now, we consider the $(y-\varepsilon)$-UF. For each component c^- of the $(y-\varepsilon)$-UF, we find the components $\{c_i^+\}$ that its members go to by performing a find query on the $(y+\varepsilon)$-UF. If there are no components that c^- flows to, then c^+ must be a local max, in which case we create a new vertex and create an edge from the vertex mapped to by c^- in the previous frontier to the new vertex (see step (8) of Figure 8). If there is only **one** component c^+ that c^- flows to, then there is no topology change, and we do nothing, as the analogous case when considering the $(y+\varepsilon)$-UF handles this situation. Otherwise, if there are **multiple** components $\{c_i^+\}$ that c^- flows to, then we are at a *fork* point. Here, we insert a vertex into the graph and add an edge from the vertex c^- maps to in the previous frontier to the new vertex. Then, for each component c^+ in $\{c_i^+\}$, we map from c^+ to the new vertex in the frontier (see step (4) of Figure 8).

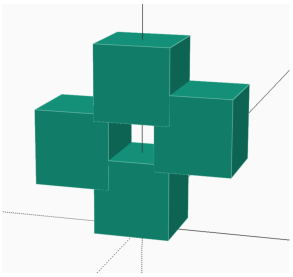
Closing The Loop. At the end of each iteration, we set the previous frontier to the current frontier, then clear the current frontier. After all iterations have finished, we will have our desired Reeb graph.

Runtime. Let n be the number of primitives in the input expression. We have a loop over the critical areas of all primitives, each of which has $\mathcal{O}(1)$ critical areas, so we have $\mathcal{O}(n)$ loop iterations. Within each loop iteration, we generate collisions for all primitives with the $(y+\varepsilon)$ and $(y-\varepsilon)$ plane, taking $\mathcal{O}(n)$ time. Then we compute the connected components of each, taking time $\mathcal{O}(\binom{n}{2}) = \mathcal{O}(n^2)$, since there are $\binom{n}{2}$ pairs of shapes to check intersections for. Thereafter, we have two loops with similar runtimes. In each loop, we iterate over each component, of which there are $\mathcal{O}(n)$, and for each component, we find the components it stems from or flows to, taking $\mathcal{O}(n)$ time. The rest of the operations within the loop are dominated by this operation. Thus, the overall runtime is $\mathcal{O}(n \cdot (2 \cdot n + 2 \cdot n^2 + 2 \cdot n^2)) = \mathcal{O}(n^3)$.

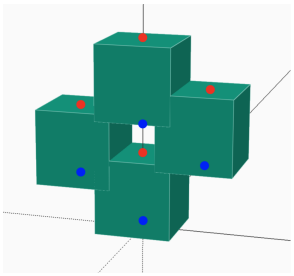
Algorithm 1 Construct Reeb Graph from CSG

Input: CSG expression e

primitives ← collect all primitives from e
critical areas ← collect and sort unique critical areas of all primitives
$\varepsilon \leftarrow$ (minimum distance between adjacent critical areas) $/2$
previous frontier $F_{\text{prev}} \leftarrow$ empty dictionary
result graph ← empty graph
for $a_i \in$ critical areas **do**
  $y_i \leftarrow \text{height}(a_i)$
  frontier $F \leftarrow$ empty dictionary
  $P^+ \leftarrow$ collisions of all primitives with plane $y = y_i + \varepsilon$
  $P^- \leftarrow$ collisions of all primitives with plane $y = y_i - \varepsilon$
  $C^+ \leftarrow$ connected components of all collisions in P^+
  $C^- \leftarrow$ connected components of all collisions in P^-
  for component $c_i^+ \in C^+$ **do**
    source components ← find components in C^- that c_i^+ stems from
    if $|\text{source components}| = 0$ **then**
      $v \leftarrow$ create vertex in graph
    else if $|\text{source components}| = 1$ **then**
      $v \leftarrow F_{\text{prev}}[\text{source component}]$
    else
      $v \leftarrow$ create vertex in graph
      for source component ∈ source components **do**
        $v' \leftarrow F_{\text{prev}}[\text{source component}]$
        add edge (v, v') to graph
      end for
    end if
    for member $m \in c_i^+$ **do**
      $F[m] \leftarrow v$
    end for
  end for
  for component $c_i^- \in C^-$ **do**
    target components ← find components in C^+ that c_i^- flows to
    if $|\text{target components}| = 0$ **then**
      $v \leftarrow$ create vertex in graph
      add edge $(F_{\text{prev}}[v], v)$
    else if $|\text{target components}| > 1$ **then**
      $v \leftarrow$ create vertex in graph
      add edge $(F_{\text{prev}}[c_i^-], v)$ to graph
      for target component ∈ target components **do**
        $F[\text{target component}] \leftarrow v$
      end for
    end if
    for member $m \in c_i^-$ **do**
      $F[m] \leftarrow v$
    end for
  end for
  previous frontier ← frontier
end for



(a) Model with torus topology



(b) Critical areas of all primitives

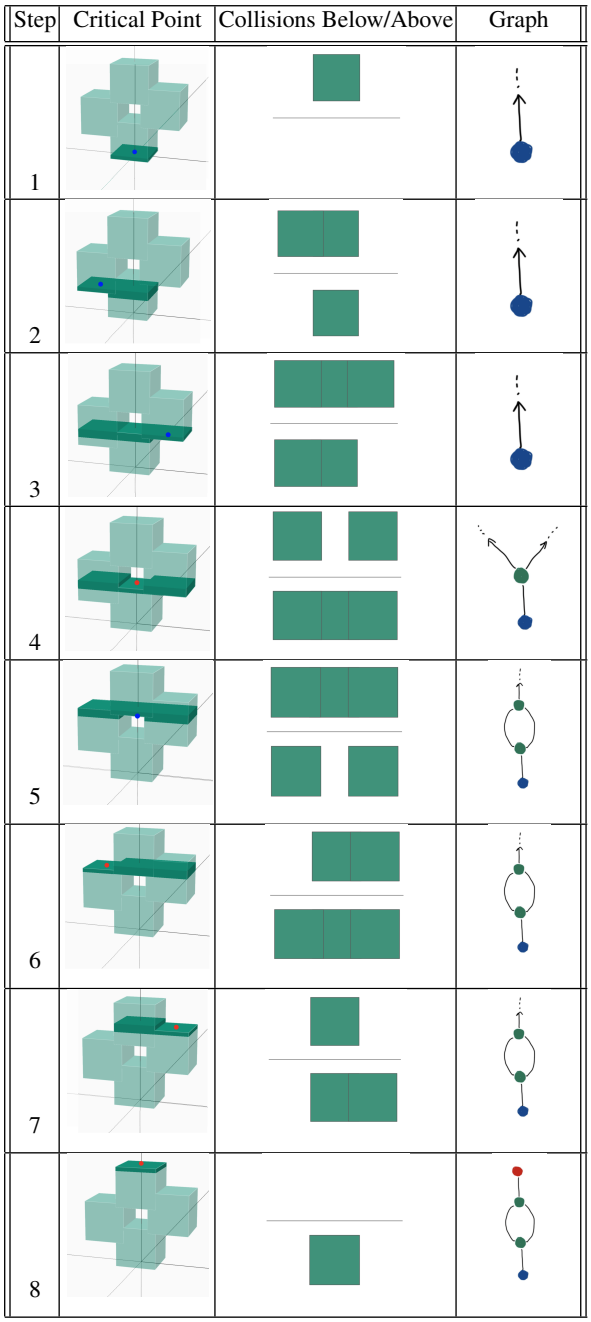


Figure 8: Execution of our algorithm on an example CSG model. We sweep upwards, beginning from the lowest critical area. At each critical area, we generate collisions below and above the critical point. To determine whether a topological change has occurred, we compare the components below and above the critical point.

5. Results

In this section, we compare the results of the ascending paths algorithm and our CSG algorithm, both of which we implemented in Julia. To this end, we crafted a set of CAD models compatible with both (i.e., restricted to cubes and the union operation) that test different cases of the CSG algorithm. To design these CAD models, we used OpenSCAD, a tool for programmatically constructing CSG models.

To generate inputs for the ascending paths algorithm, we exported the models to `.off` files and loaded them into Julia via MeshIO.jl. To generate inputs for our CSG algorithm, we transcribed the abstract syntax trees manually from OpenSCAD to Julia.

The results of this process are shown in Figure 9. The initial goal of these comparisons was to build confidence in the correctness of our CSG algorithm, and indeed, on most of the models, the two algorithms agree. However, there is one model in which the two algorithms differed. In the last model in Figure 9, the ascending paths algorithm produces more saddle point nodes (the blue nodes) than the CSG algorithm. If we analyze the level sets of this model, we find there are three components below the first pair of saddle nodes and only one component above. A faithful encoding of this topological change then is given by the CSG result, which has an in-degree of 3 and an out-degree of 1. That is, the ascending paths algorithm produces the incorrect result here!

Note that while the ascending paths algorithm returns the incorrect result in this case, the algorithm itself is not flawed, as it makes no promises in this case. In particular, the algorithm presupposes that each critical point has a distinct critical value. For the height function, this means each critical point must exist at a distinct altitude. Under this assumption, each node in the Reeb graph can have a maximum in- and out-degree of 2.

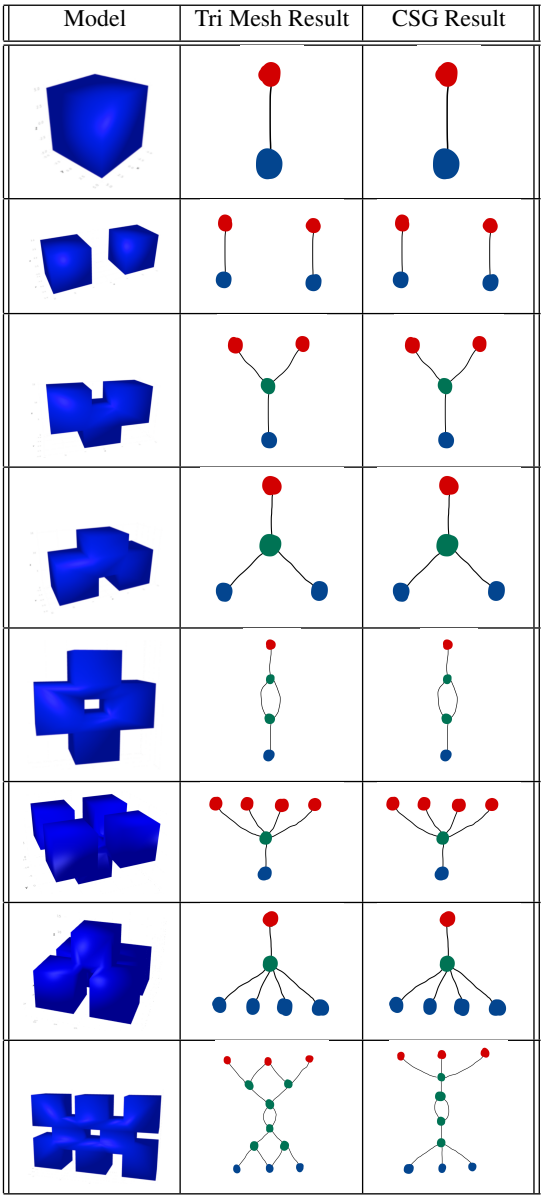


Figure 9: Collection of models and the corresponding Reeb graphs produced by each of the algorithms we implemented.

6. Discussion

Proof of Correctness. The most obvious component missing from the presentation of the CSG algorithm is a proof of correctness! Unfortunately, the author did not provide himself ample time to explore this facet of the project in depth. The most interesting part of the proof would likely be demonstrating that differences in connected components at $(y-\varepsilon)$ and $(y+\varepsilon)$ implies a corresponding change in topology.

Intrinsic vs. Extrinsic Functions. Focusing on Reeb graphs of *height functions* makes sense from an algorithmic perspective, as it lends itself to "sweep"-style algorithms. However, from a theoretical perspective, the height function is less compelling, as it is dependent upon the surface's immersion into $\mathbb{R}^3$ (i.e., it is an *extrinsic* function). There may be more interesting algorithmic opportunities if we choose an *intrinsic* function (e.g., geodesic distance), though the author has not considered this approach in depth.

Pruning Comparisons. There's interesting future work in faster computation of the connected components at each y-level. Currently, we check all $\binom{n}{2}$ pairs of shapes for intersections, but since we're maintaining disjoint sets, there are likely some comparisons we can prune at each step, which could drastically reduce the number of comparisons.

Incremental Computation. We are currently recomputing the intersections with the plane at every critical area, but we have enough information to compute the topology changes incrementally. For example, at each critical area, we know not just that it is a critical area, but we know whether it is a minimum/maximum, **and** we know which shape it belongs to. Thus, we could localize our reasoning to the shape that corresponds to each event and determine whether it introduces a new component, removes a component, splits a component, or joins a component. The most curious

of these events is *splitting* a component, because this operation is unsupported in the classical union-find data structure, so we would likely need to augment it. The original work of Shinagawa, Kunii, and Kergosien [SKK91] use a tree-like data structure to encode the events they consider, and the present author has yet to fully investigate whether their work would be applicable here. This type of formulation falls more cleanly into the class of "sweeping" algorithms, and would bring significant efficiency benefits.

7. Acknowledgements

The author wishes to thank David Palmer for his constructive feedback on initial iterations of this project.

References

[BB13] BARRA V., BIASOTTI S.: 3d shape retrieval using kernels on extended reeb graphs. *Pattern Recognit. 46* (2013), 2985–2999. 3

[BFL16] BAUER U., FABIO B. D., LANDI C.: An edit distance for reeb graphs. In *3DOR@Eurographics* (2016). 3

[BFS00] BIASOTTI S., FALCIDIENO B., SPAGNUOLO M.: Extended reeb graphs for surface understanding and description. In *DGCI* (2000). 2, 3

[BGSF08] BIASOTTI S., GIORGI D., SPAGNUOLO M., FALCIDIENO B.: Reeb graphs for shape analysis and applications. *Theor. Comput. Sci. 392* (2008), 5–22. 1, 3

[BGW14] BAUER U., GE X., WANG Y.: Measuring distance between reeb graphs. *Proceedings of the thirtieth annual symposium on Computational geometry* (2014). 3

[BLM20] BAUER U., LANDI C., MÉMOLI F.: The reeb graph edit distance is universal. In *Symposium on Computational Geometry* (2020). 3

[BMSF06] BIASOTTI S., MARINI S., SPAGNUOLO M., FALCIDIENO B.: Sub-part correspondence by structural descriptors of 3d shapes. *Computer-Aided Design 38*, 9 (2006), 1002–1019. Shape Similarity Detection and Search for CAD/CAE Applications. URL: https://www.sciencedirect.com/science/article/pii/S0010448506001345, doi:https://doi.org/10.1016/j.cad.2006.07.003. 3

[BMW15] BAUER U., MUNCH E., WANG Y.: Strong equivalence of the interleaving and functional distortion metrics for reeb graphs. *ArXiv abs/1412.6646* (2015). 3

[BRS03] BESPALOV D., REGLI W. C., SHOKOUFANDEH A.: Reeb graph based shape retrieval for cad. 1, 3

[DN09] DORAISWAMY H., NATARAJAN V.: Efficient algorithms for computing reeb graphs. *Comput. Geom. 42* (2009), 606–616. 3

[DN13] DORAISWAMY H., NATARAJAN V.: Computing reeb graphs as a union of contour trees. *IEEE Transactions on Visualization and Computer Graphics 19* (2013), 249–262. 3

[FL16] FABIO B. D., LANDI C.: The edit distance for reeb graphs of surfaces. *Discrete & Computational Geometry 55* (2016), 423–461. 3

[HR20] HAJIJ M., ROSEN P.: An efficient data retrieval parallel reeb graph algorithm. *ArXiv abs/1810.08310* (2020). 2, 3, 4

[HSKK01] HILAGA M., SHINAGAWA Y., KOMURA T., KUNII T.: Topology matching for fully automatic similarity estimation of 3d shapes. *Proceedings of the 28th annual conference on Computer graphics and interactive techniques* (2001). 1, 2

[HWW10] HARVEY W., WANG Y., WENGER R.: A randomized o(m log m) time algorithm for computing reeb graphs of arbitrary simplicial complexes. *Proceedings of the twenty-sixth annual symposium on Computational geometry* (2010). 3

[Mun17] MUNCH E.: A user's guide to topological data analysis. *Journal of learning Analytics 4* (2017), 47–61. 1, 3

[PCM03] PASCUCCI V., COLE-MCLAUGHLIN K.: Parallel computation of the topology of level sets. *Algorithmica 38* (2003), 249–268. 3

[PSBM07] PASCUCCI V., SCORZELLI G., BREMER P., MASCARENHAS A.: Robust on-line computation of reeb graphs: simplicity and speed. In *SIGGRAPH 2007* (2007). 3

[Ree46] REEB G.: Sur les points singuliers d'une forme de pfaff complètement intégrable ou d'une fonction numérique. *C. R. Acad. Sci. Paris 222* (1946), 847–849. 1, 2

[SKK91] SHINAGAWA Y., KUNII T., KERGOSIEN Y.: Surface coding based on morse theory. *IEEE Computer Graphics and Applications 11*, 5 (1991), 66–78. doi:10.1109/38.90568. 1, 2, 3, 8

[SMP16] SILVA V. D., MUNCH E., PATEL A.: Categorified reeb graphs. *Discrete & Computational Geometry 55* (2016), 854–906. 3

[TFL*18] TIERNY J., FAVELIER G., LEVINE J. A., GUEUNET C., MICHAUX M.: The topology toolkit. *IEEE Transactions on Visualization and Computer Graphics 24* (2018), 832–842. 3