Implementation of a Triangulation Data Structure

Etta Rapp, Elisheva Sigfried

Topics in Topology: Computational Topology

Spring 2020

Abstract

In this paper, we describe our implementation of a data structure to store a triangulation of a compact, connected two-manifold without boundary. We provide mathematical background related to surfaces, triangulations, and symmetry groups of triangles. We then discuss the data structure, its uses, and our design choices in implementing it.

Introduction

A triangulation of a surface is a simplicial complex homeomorphic to the surface. This is a topological representation of the surface that can be stored in a computer as a data structure and can then be used to determine various properties about the given surface.

Background

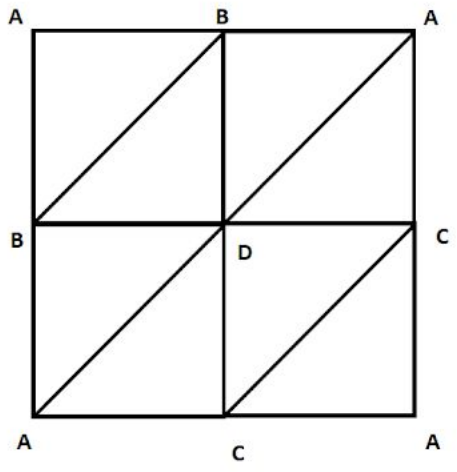
A surface or 2-manifold[1] without boundary is a two-dimensional topological space $\mathbf{M}$ such that every point in $\mathbf{M}$ is a member of an open set that is homeomorphic (topologically equivalent) to the unit disc.[2] We can then say the manifold is compact if the topological space $\mathbf{M}$ is compact, meaning that every open cover of $\mathbf{M}$ has a finite subcover.

[1] We will be using the two terms interchangeably throughout this paper.

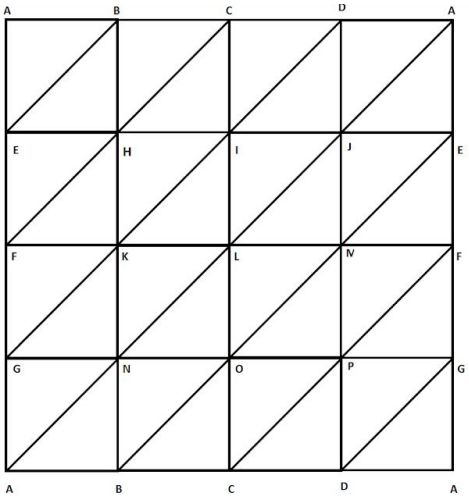
[2] More precisely, if we let $D = \{x \in \Re^2 : ||x| < 1|\}$ be the open unit disc then a 2-Manifold is the topological space $\mathbf{M}$ such that $\forall x \in \mathbf{M} \exists\ V \subseteq M$ open set with $x \in V$ such that V is homeomorphic to D (i.e there are topologically equivalent).

Additionally, the manifold is called connected if the topological space **M** is connected, meaning there is no pair of non-empty open sets U and V whose intersection is the null set and whose union is **M**.

In order to better understand a surface we can triangulate it by breaking it up into regions that are homeomorphic to triangles. The resulting triangulation is a union of triangles. Each triangle in a valid triangulation must have distinct edges and distinct vertices. Additionally, all triangles of a triangulation must either be disjoint, share one edge, or share one vertex. For example, Figure 1 appears to be a valid triangulation of a sphere, but many of the triangles do not have distinct edges and vertices, and there are triangles that have more than one edge in common, rendering this triangulation invalid. Figure 2, on the other hand, is a valid triangulation of a torus since all of its triangles are either disjoint or share only one edge or one vertex.

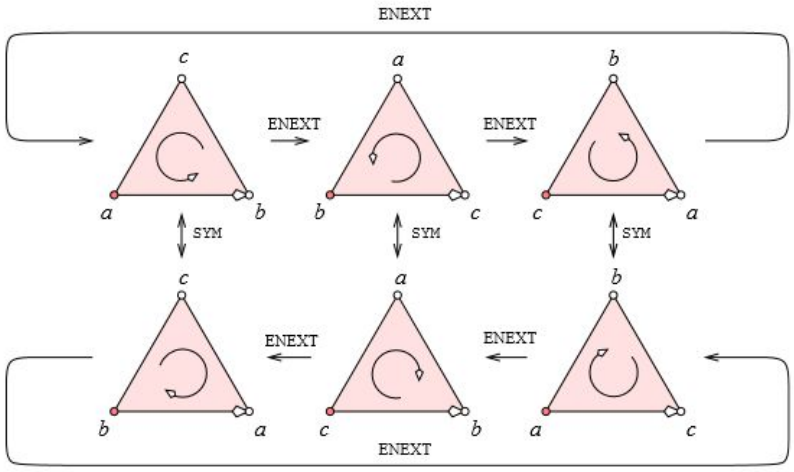


(Figure 1: invalid triangulation of sphere)



(Figure 2: valid triangulation of torus)

A key element of a data structure to represent a triangulation is the symmetry group of a triangle. A triangle's symmetry group contains six triangles with the same vertices as the triangle, each with a different ordering of the vertices. These six triangles are called ordered triangles, and are obtained by applying cyclic shifts (advancing the lead vertex of the triangle) and transposition (reversing the lead edge of the triangle) to the triangle. For example, given a triangle with vertices a, b, and c, the six ordered triangles generated would be abc, bca, cab, bac, cba, and acb, as in Figure 3. In this example abc, bca, and cab have the same orientation while bac, cba, and acb have the reverse orientation.



(Figure 3: The symmetry group of a triangle with vertices a, b, and c. ENEXT arrows represent advancing the lead edge; SYM arrows represent reversing the lead edge.)

Implementation of Data Structure

We implement a data structure to store a triangulation of a compact, connected 2-manifold without boundary as a hierarchy of nested classes. The outermost class in our structure is a Triangulation class whose attributes are a set of vertices and a Graph object. We store the vertices in a set rather than a list in order to avoid duplicates and increase efficiency.

Graph, an inner class of Triangulation, contains a dictionary in which the keys are labels called mu and the values are the corresponding Triangle objects which are the nodes of the Graph. The dictionary enables us to easily and efficiently access Triangles in our triangulation.

A Triangle object has a label mu, a list of its three vertices, a list of its three neighboring Triangle objects (the triangles that share an edge with this Triangle), and an attribute called symGroups (a list of the six OrderedTriangle objects that make up the symmetry group of this Triangle). Arcs of a Graph are stored implicitly within Triangle objects; two Triangle objects that are connected by an arc are in each other's lists of neighbors. When a Triangle object is initialized, it creates its label mu by alphabetically sorting and then concatenating the characters that refer to its three vertices.

An OrderedTriangle object has the same label mu as the Triangle that it is contained in and a variable called i that identifies the ordering of its vertices. The i values 0, 1, 2, 4, 5, 6 correspond to permutations abc, bca, cab, bac, cba, and acb respectively. An OrderedTriangle also has a variable called org that identifies its lead vertex and a variable fnext (not implemented in our code) that points to the OrderedTriangle in a neighboring Triangle that shares a lead edge with this OrderedTriangle.

A user enters a triangulation into our data structure as a series of triangles, each identified as a set of three vertices, which are passed to the addTriangle method of the Triangulation

class. We assume that the user enters a valid triangulation. This method adds the three nodes to the Triangulation's set of vertices and adds them to the Triangulation's Graph object using the Graph's addNode method. The addNode method then creates a Triangle object from the three vertices it receives as parameters, adds the Triangle object's neighbors to its list of neighbors appropriately, and appends the Triangle with its label to the Graph's dictionary of nodes.

Neighboring triangles are identified and added using two methods of the Triangle class, sharesEdgeWith to check whether two Triangle objects share an edge, and addArc to create an arc between them. The sharesEdgeWith method takes two Triangle objects as parameters and counts up how many vertices they have in common, returning a boolean value identifying whether the number of shared vertices is equal to two. The addArc method is then called twice to add each Triangle object to the list of neighbors of the Triangle object that it is connected to.

We added several other methods to our classes that are useful in applications of triangulations. To our OrderedTriangle class we added an ENEXT method that returns the OrderedTriangle created by advancing its lead edge and a SYM method that returns the OrderedTriangle created by reversing the direction of the lead edge (see Figure 3.) ENEXT is calculated by manipulating the i value of an OrderedTriangle, returning $((i + 1) \bmod 3)$ if i is less than or equal to two and $((i+1) \bmod 3 + 4)$ otherwise. SYM similarly manipulates i, returning $((i + 4) \bmod 8)$.

We added methods to our Triangulation class to return a few statistics describing the Triangulation: the number of vertices, the number of faces, the number of edges, the Euler characteristic, and the genus. The Euler characteristic, denoted χ, is computed as the number of vertices of a triangulation minus the number of edges plus the number of faces. It can be used to calculate genus, the number of two-dimensional holes in the triangulated manifold, as $\chi = 2 - 2g$ where g is the genus of the surface. This genus method only works for orientable surfaces; there is a different equation for the genus of a non-orientable surface.

Our data structure also includes a simple Depth-first Search algorithm that recursively traverses our data structure. Our implementation of the search counts the number of triangles in the triangulation, but it can easily be adapted to traverse the triangulation for other purposes. For example, a Depth-first Search is used in determining the orientability of a manifold.

Computer Experiments/Simulations and Results

We tested our code by applying it to a sphere and a torus. We triangulate a sphere as a tetrahedron and create a corresponding Triangulation object. Our test code then calculates the Euler characteristic of the sphere and correctly returns that the genus of a sphere is zero. Our torus test code creates a Triangulation object for a torus as in Figure 2 and adds all of the triangles in the figure to the triangulation. It then calculates the Euler

characteristic and the genus of the torus. Our code accurately returns that the genus of a torus is one, confirming that our implementation of the data structure is accurate.

Conclusion

A triangulation of a surface has many uses in topology. Using a data structure to store a triangulation of a surface expands our capabilities and enables us to study and compute properties of surfaces that are difficult to visualize. We can then perform computations that would be difficult to do by hand. Once a triangulation is stored in a data structure, we can easily extend it with other methods including algorithms to traverse a triangulation, determine the orientability of a surface, and classify a manifold.

References cited

"Chapter II.1, II.2." *Computational Topology: an Introduction*, by Herbert Edelsbrunner and John Harer, American Mathematical Society, 2010, pp. 27–35.

Gidea, Marian. "Topics in Topology". Stern College, Yeshiva University. Spring 2020. Lecture notes and lecture.

The following Wikipedia pages were also used to clarify some ideas and definitions:

https://en.wikipedia.org/wiki/Manifold

https://en.wikipedia.org/wiki/Triangulation_(topology)

https://en.wikipedia.org/wiki/Surface_triangulation

https://en.wikipedia.org/wiki/Compact_space