# ‘Canonical’ Form and Faster Algorithm of Ribbon Graph

Xiaxin Li[*] Zelin Lv[†] Kexiang Wang[‡]

**Abstract**

This paper mainly deals with problems of improving the existing code of Riccado Murri [1]. By introducing a new concept: ‘Canonical form’ of ribbon graphs, we manage to reducing the time of comparing isomorphisms between 2 ribbon graphs. Using this way, we may get homology of moduli spaces of curves with higher genus g and components n (i.e. $M_{0,7}$, $M_{2,3}$)[1].

## 1 Introduction

In this paper, we present an improved algorithm based on the ideas given by Riccardo Murri of computing homology of moduli spaces of curves [1]. Our method first generates the family of ‘Canonical’ forms of one given ribbon graph and then store a pair of them using dictionary, which is a data structure based on Hash, where the key is the generated ‘Canonical’ form and its corresponding value is the original fatgraph. By our algorithm, two generated set of ”Canonical” forms are identical if and only if the original fatgraphs are isomorphic. Using this property and constant time property of dictionary. We reduce the time required for comparing if two given fatgraphs are isomorphic so therefore efficiently improved the speed of computing homology of moduli spaces of curves with higher genus $g$ and number of component $n$.

We first introduce how Riccardo Murri algorithm functions to compute the homology of moduli spaces of curves. Since we only revise his code in comparing isomorphism, we will just briefly describe why we can get the homology by computing the rank and nullity of vector spaces composed by ribbon graphs. We will focus more on how he implements code to generate ribbon graphs for fixed g and n and comparing isomorphisms of contracted graphs.

Then we introduce how our method improve the speed of computing homology of moduli spaces of curves. First, we give an algorithm to generate the family of ‘Canonical’

[*]University of Wisconsin–Madison, Madison, WI, USA, xli685@wisc.edu
[†]University of Wisconsin–Madison, Madison, WI, USA, zlv7@wisc.eduu
[‡]University of Wisconsin–Madison, Madison, WI, USA, kwang365@wisc.edu
[1]Our code can be found at `https://github.com/zelinlv/Faster-Algorithm-of-RibbonGraph`

forms of one given ribbon graph. Note that ehe set of 'Canonical forms' are identical if and only if the original fatgraphs are isomorphic. And the size of such a set is relatively small comparing to all isomorphic ways to represent a graph. In order to construct 'Canonical' forms of a ribbon graph, we uniquely traverse the given graph. Note that our way of representing the 'Canonical forms' of a ribbon graph is using 'half edge' and making it become a number string in a list. We will prove later in this paper that this way of representing a ribbon graph is unique up to isomorphism.

At last, we will make a table to compare the time of his way to compute homology of moduli spaces of curves for give g and n with ours.

## 2 Summary of how to compute isomorphism of ribbon graph objects

This section summarizes how Riccardo Murri represents ribbon graphs and how he computes isomorphism of ribbon graph objects.

### 2.1 Ribbon Graphs and its representation

The formal definitions and related context can be found in [2]. Here we just briefly describe the definition related to our algorithm.

**Definition 1. (Geometric denition of ribbon graphs)**: A ribbon graph is a finite CW complex of pure dimension 1, together with an assignment, for each vertex $v$, of a cyclic ordering of the edges incident at $v$.

Based on this geometric definition, we can represent ribbon graph objects in computer by the following way:

**Definition 2.** A Ribbon graph object $G$ is comprised of the following attributes:

- A list $G.vertices$ of Vertex objects;
- A list $G.edges$ of Edge objects;
- A set $G.boundary_cycles$ of Boundary Cycle objects;
- An orientation $G.orient$.

**Definition 3.** A Vertex object $v = Vertex(e_1, ..., e_z)$ is a list of the labels $e_1, ..., e_z$ of attached edges. Two Vertex objects are considered equal if one is equal (as a sequence) to the other rotated by a certain amount.

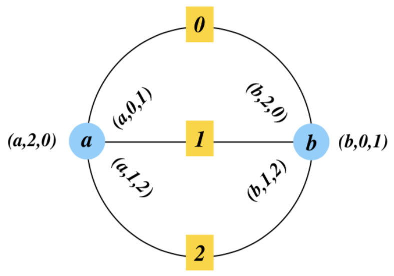
**Definition 4.** An Edge object e is an unordered pair of endpoints, so we define: each endpoint corresponds to a 2-tuple (v,a), where v is a vertex, and a is the index at which edge e appears within vertex v (the attachment index).

**Definition 5.** A BoundaryCycle object is a set of corners (see Figure 3.3).

A corner object $C$ is a triple (vertex, incoming, outgoing), consisting of a vertex $v$ and two indices $i = C.incoming$, $j = C.outgoing$ of consecutive edges (in the cyclic order at $v$). In order to have a unique representation of any corner, we impose the condition that either $j = i + 1$, or $i$ and $j$ are, respectively, the ending and starting indices of $v$ (regarded as a list).

**Definition 6.** The Orientation $G.orient$ is a list that associates each edge with its position according to the order given by the orientation. Two such lists are equivalent if they differ by an even permutation.

**Here is an example:**



In this figure, we know that the list of vertices is $G.vertices = [a(0, 1, 2), b(0, 2, 1)]$. The list of edges are represented as $G.edges = [edge0(a, 0), (b, 0); edge1(a, 1), (b, 1); edge2(a, 2), (b, 2)]$ the boundary cycle can be represented as $G.boundarycycles = [(a, 0, 1); (b, 2, 0); (a, 1, 2); (b, 1, 2)]$. Note that the index of boundary cycle is not necessary to be the index of edges. Here, we can pick a vertices and choose a starting point as 0 and label the edges clock-wisely. For example, for vertex b, we have that 0 is the starting edge, but according to clockwise order, the next edge is edge 2 not 1.

Last, there are many orientations of this graph. Basically, any orientation of edges can be seen as an orientation.

## 2.2 Ribbon graph isomorphism definition

**Definition 7. (Isomorphisms between 2 Ribbon Graphs)**
An isomorphism of 2 ribbon graphs $G_1$ and $G_2$ can be represented as a map $f = (pv, rot, pe)$ where:

- $pv$ is a permutation of the vertices: vertex $v_1$ of $G_1$ is sent to vertex $pv[v]$ of $G_2$, and rotated by $rot[v]$ places leftwards;
- $pe$ is permutation of the edge labels: edge $e$ in $G_1$ is mapped to edge $pe[e]$ in $G_2$.

The isomorphisms of ribbon graphs are similar to normal graphs. When we want to check isomorphisms between 2 graphs, we just need to check they are matched with respect to edges and vertices.

### 2.3 How to compare isomophisms between ribbon graphs

In Murri's algorithm, when he generates all the ribbon graphs of a moduli space of genus g with n marked points, he creates a list storing all non-isomorphic ribbon graph objects. Every time a new ribbon graph is generated, he compares whether this new graph is isomorphic to any of the graphs already in the list. If there exists an isomorphism, then discard this new graph, otherwise add this graph into the list. For example, if there are a thousand non-isomorphic ribbon graphs in the list, and a newly generated ribbon graph is yet to be added into the list. Murri's algorithm will induce a thousand times of comparing isomorphism, which is very time-consuming.

## 3 Cononical Form Generation and Improved Algorithm

In our new algorithm, when we create a list of all non-isomorphic ribbon graphs of a certain $M_{g,n}$, we do not compare isomorphism at all. Instead, we calculate the canonical forms (which will be defined below) of the ribbon graphs and store them in separate lists. The detailed process is described after giving some definitions:

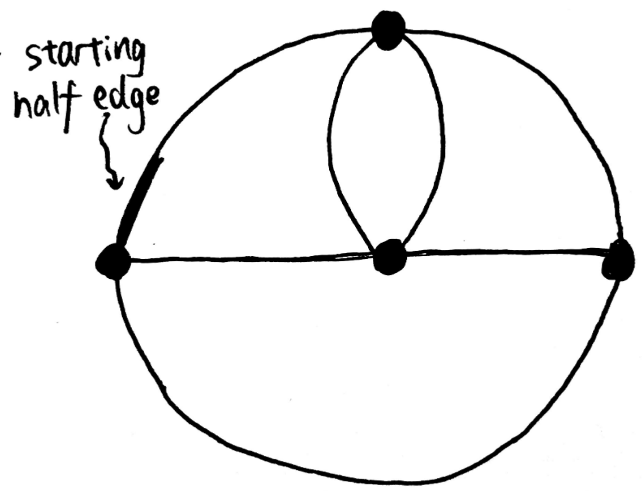
**Definition 8. Canonical form of a ribbon graph** : A canonical form of a ribbon graph is an object that is produced by picking a staring half edge and then traverse the ribbon graph by X order (defined below).

Note that, canonical form object is not a ribbon graph object (in programming, they are different objects, although they are both lists of lists), but up to isomorphism, one canonical form object uniquely represents a ribbon graph object.

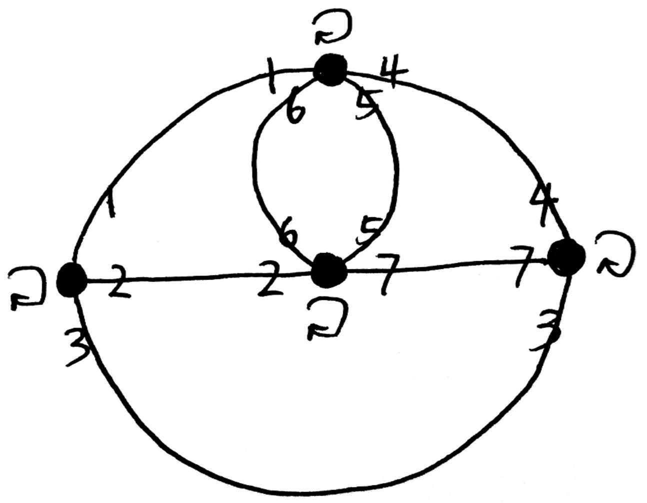
**Definition 9. X order**: The X order is a way of traversing a ribbon graph by first traversing over all cyclic neighbors of the half edge in order, then traverse the neighbor of the half edge.

As what will illustrate in the examples below, traversing a ribbon graph in X order is analogous to traversing the ribbon graph by using BFS (Breadth first search).

For example, if we traverse the following unmarked ribbon graph G, starting from the half edge marked below,



we will get the following canonical form of this graph, name it CF, as shown below,



As a consequence, the canonical form of $G$ is $CF[[1, 2, 3], [1, 4, 5, 6], [2, 6, 5, 7], [3, 7, 4]]$.

One important thing to notice is that a single ribbon graph usually have more than one canonical forms. That is, if we start on a different half edge, we may get different canonical forms representing the same ribbon graph. Hence, we define the two following terms:

**Definition 10. All canonical forms of a ribbon graph**: All canonical forms of a ribbon graph is a collection of all possible canonical forms of this ribbon graph.

**Definition 11. Representative canonical form of a ribbon graph**: A representative canonical form of a ribbon graph is one of the canonical forms of the ribbon graph, randomly picked from the collection of all canonical forms of the ribbon graph.

Now we describe the detailed procedures of our new algorithm of generating all non-isomorphic ribbon graphs of a certain g and n.

Denote ARCFList (namely, all representative canonical forms list) as the list which stores the representative canonical forms of all non-isomorphic ribbon graphs of given

$g$ and $n$. Based on the fact that, up to isomorphism, one representative canonical form represents a unique ribbon graph, When we want to add a new ribbon graph $G$ in is a ribbon graph $G$, we first generates all of its canonical forms and store them in a list, name it GList, then we take each of the element in this GList and compare whether this element is exactly equal to an element in the ARCFList (namely, all canonical form list, which stores all representative canonical forms). If there is equality, it means that this ribbon graph is already in the list so we discard it. If not, then pick a representative canonical form of this graph in GList and add it to ARCFList. At the same time, add this graph object into AllList, which stores all non-isomorphic ribbon graphs of type $(g, n)$.

A key distinction to notice here is that AllList stores all non-isomorphic ribbon graph objects, while ARCFList stores all representative canonical forms, which are not ribbon graph objects. In this way, lots of time can be saved because it compares equality rather than isomorphism, and thus isomorphic ribbon graph objects can be determined much quicker: one equality implies discard.

More examples related to canonical forms are given below.

**Example 1**: The canonical form of $Fatgraph[[0, 1, 2], [0, 2, 1]]$ object, if we start at the $(0, 0)$ half edge, which corresponds to the first vertex and its first entry 0, we will get the canonical form $CF[[1, 2, 3], [1, 3, 2]]$.

**Example 2**: (Restatement of a previous example) The canonical form of $Fatgraph[[6, 5, 4, 0], [5, 3, 1, 4], [2, 3, 6], [1, 2, 0]]$ object, if we start at the $(3, 2)$ half edge which corresponds to the third vertex and its second entry (the entry is 0), we will get the canonical form $CF[[1, 2, 3], [1, 4, 5, 6], [2, 6, 5, 7], [3, 7, 4]]$.

**Example 3**:: All the canonical forms of $Fatgraph[[6, 2, 7, 4], [3, 7, 1], [2, 5, 6, 0, 1, 0], [4, 3, 5]]$ object are:

- $CF1[[1, 2, 3, 4, 5, 4], [1, 6, 7, 3], [2, 7, 8], [5, 8, 6]]$
- $CF2[[1, 2, 3, 4, 3, 5], [1, 6, 7], [2, 5, 8, 6], [4, 7, 8]]$
- $CF3[[1, 2, 3, 2, 4, 5], [1, 4, 6, 7], [3, 8, 6], [5, 7, 8]]$
- $CF4[[1, 2, 1, 3, 4, 5], [2, 6, 7], [3, 7, 8, 5], [4, 8, 6]]$
- $CF5[[1, 2, 3, 4, 5, 2], [1, 6, 7], [3, 7, 8, 5], [4, 8, 6]]$
- $CF6[[1, 2, 3, 4, 1, 5], [2, 6, 7, 4], [3, 7, 8], [5, 8, 6]]$

**Algorithm 1** BFS(fg, half Edge, canonical form, q, i half e, visited, v, w, corres, dfneigh, finish)

```
$v_{index} := halfEdge[0]$
$e_{index} := halfEdge[1]$
$finish := finish + 1$
if the neighbor of this half-Edge is visited previously: then
  label this half-Edge the same number as that of its neighbor
else
  i half e := i half e + 1
  $canonical\ form[v][e_{index}] := i\ half\ e$
end if
if the number of visited half edge is greater than the number of half edge in the
graph: then
  $return\ canonical\ form$
end if
for edge in all cyclic neighbors of this half Edge: do
  if edge is not in visited: then
    q.put(edge)
  end if
end for
if the neighbor of this half-Edge is not in visited: then
  (Denote the neighbor as edge neighbor)
  dfneigh[edge neighbor] := 1
  (The dfneigh array denotes a 1 if the corresponding edge is the neighbor of previous
  visited edge, otherwise 0)
  q.put(edge neighbor)
end if
next half edge := q.get()
while next half edge is in visited and q is not empty: do
  next half edge := q.get()
end while
if next half edge[0] not in corres: then
  (corres is a list linking the index of vertices in fg and vertices in canonical form)
  Append a new empty vertex into the canonical form with the right length
  w := w+1
  v := w
  corres[v] := next half edge[0]
else
  if next half edge[0] not equal to half Edge[0]: then
    change v to the right index
  end if
end if
return BFS(fg, next half edge, canonical form, q, i half e, visited, v, w, corres,
dfneigh, finish)
```

## 4 Experiment

| | Total Running Time (s) obtained by [1]: | Total Running Time (s) obtained by our code: |
|---|---|---|
| $M_{0,3}$ | 0.12 | 0.43 |
| $M_{0,4}$ | 0.29 | 0.85 |
| $M_{0,5}$ | 29.43 | 23.01 |
| $M_{1,1}$ | 0.128 | 0.42 |
| $M_{1,2}$ | 0.27 | 0.43 |
| $M_{1,3}$ | 174.75 | 102.89 |
| $M_{2,1}$ | 7.39 | 6.74 |
| $M_{2,2}$ | 224694.61 | around 120000 |

Table 1: Total CPU time (seconds) used by the Betti numbers computation for the indicated $M_{g,n}$ spaces. Riccardo's Running time was sampled on the `idhydra.uzh.ch` computer of the University of Zurich, equipped with 480GB of RAM and Intel Xeon CPUs model X7542 running at 2.67GHz; Python version 2.6.0 installed on the SUSE Linux Enterprise Server 11 64-bits operating system was used to execute the program. Our running time was sample on the `http://math.wisc.edu/` computer of University of Wisconsin, Madison, equipped with Intel(R) Xeon(R) CPU E5-2450 running at 2.10GHz; Linux Version 18.10 and Python Version 2.7.15+.

By comparing the results, we can see our improved algorithm has gained significant speedup when calculating the complicated cases.

## 5 Conclusion and Future Direction

In this paper, we showed that our method is effective by conducting the experiment of comparing the running time of our code and original code. Our algorithm has much better performance when the required computation resource are large. Intuitively, when reduce the running time of the modified part of original code, using asymptotic analysis, from $O(n^2)$ to $O(n)$, where $n$ is the size of given graphs. Therefore, the improvement gets more significant when the size gets larger.

The next step, with the improved speed, would be calculating the homology of moduli spaces of curves with higher genus $g$ and number of component $n$, such $g = 2, n = 3$. In order to further reduce the time required in computing homology, parallelism could be employed on this program. This program has a natural way to be paralleled that the last part of it to construct a matrix by a series of constructing smaller matrix can be easily paralleled. Therefore, it could be used to have some new results.

**Acknowledgments.** This is the final project of Zelin Lv, Xiaxin Li and Kexiang Wang for Math 490 Curl (Collaborative Undergraduate Research Lab) under guidance of Professor Andrei Caldararu at University of Wisconsin-Madison.

## References

[1] Riccardo Murri. Fatgraph Algorithms and the Homology of the Kontsevich Complex Pre-print available at: https://arxiv.org/abs/1202.1820

[2] Robert Clark Penner, Michael Knudsen, Carsten Wiuf, and Joergen Ellegaard Andersen. Fatgraph Models of Protein. *Proteins*, 227:32, 2009.