Getting GBM Order for Graphs

Aryaman Maithani

December 16, 2024

Contents

Introduction

This document is meant to be a documentation for a program that accompanies the paper [CHM24]. The goal of the program is to take as input a graph G and return an order $\prec$ on the set of edges of G, such that the corresponding generalized Barile-Macchia resolution is minimal. We shall refer to such an order as a *good* order. This document assumes familiarity with the concepts mentioned in that paper.

The program is written using the computer algebra system `SageMath` [The23]. Part I describes two different ways that the reader can run this program. For the reader unfamiliar with `SageMath` and `python`, Section 1 will likely be all they want to read from this part of the documentation.

Part II describes how the program works. In particular, Section 6 describes the graphs for which the program will return an order.

In the entirety of this documentation, a *graph* will always be finite and simple. We use the abbreviation "BM" for "Barile-Macchia" and "GBM" for "generalized Barile-Macchia".

Part I

How to use

1 Running it online (without SageMath installation)

This section assumes no familiarity with SageMath and describes a way to run the program without having SageMath installed on your local machine. Instead, we make use of the cloud-based collaborative software CoCalc [Sag20], which gives an online interface to run SageMath.

STEP 0. Go to cocalc.com and create an account.

STEP 1. Download the file GBM_order.tar.gz from
aryamanmaithani.github.io/research_codes/GBM/GBM_order.tar.gz.

STEP 2. Create a new project on CoCalc.

STEP 3. Upload GBM_order.tar.gz to the project.

STEP 4. Open the file GBM_order.tar.gz in the project. Click on Extract Files... once.

STEP 5. Return back to the project (using the Explorer button on the left).

STEP 6. Click on the newly created folder GBM_order.

STEP 7. Open the file interface.ipynb. This will open an IPython Notebook that runs the program. Further instructions are mentioned in the notebook.

Remark 1.1. The output of the program will be a list of edges. These edges are listed in ascending order. That is, if the output $[e_1, \ldots, e_n]$, then the order is $e_1 \prec \cdots \prec e_n$. $\triangleleft$

If the reader is familiar with SageMath or python, then Sections 3 and 4 might be helpful as well.

2 Running it locally

In this section, we assume that the user has SageMath installed on their local machine. In particular, we assume that running the command

```
$ sage my_file.py
```

in the terminal executes the file my_file.py.

STEP 1. Download the file GBM_order.tar.gz from
aryamanmaithani.github.io/research_codes/GBM/GBM_order.tar.gz.

STEP 2. Extract the files into a folder of your choice.

STEP 3. Start the terminal from the above folder.

STEP 4. Run the command

```
$ sage interface.py
```

in the terminal.

STEP 5. You will be shown an "Enter graph:" prompt. Enter any valid input using which SageMath's Graph command can create a graph, and press the Enter/return key. See Section 3 for some common ways of inputting a graph.

The relevant response will then be printed. See Remark 1.1.

3 Ways of defining a graph

In this section, we describe different ways of giving an input to the "`Enter graph:`" prompt.

> For the programmer: The input given is stored as a string in a variable called `graph_input`. Then, the graph is created using `G = Graph(eval(graph_input))`.

We now describe some of the various ways of inputting a graph.

(a) Enter the graph as a list of edges. Example:
`[(0, 1), (0, 2), (0, 3), (0, 4)]`
This creates a graph with five vertices: $0, \ldots, 4$ and four edges as listed above.

(b) Enter the vertices are their neighbours as a dictionary. Example:
`{1 : [2, 3, 4, 5], 2 : [1, 3], 3 : [1, 2], 4 : [5]}`
The creates a graph with five vertices: $1, \ldots, 5$. Moreover, it creates an edge between two vertices whenever one is listed as a neighbour of the other.
(The above dictionary need not be "symmetric". For example, 4 is listed a neighbour of 1 but 1 is not listed as a neighbour of 4. Regardless, the edge `(1, 4)` will be present.)

(c) The Sage library has some commands for generating common graphs. Examples:

```
graphs.PathGraph(6)
graphs.CycleGraph(5)
graphs.CompleteGraph(6)
graphs.CompleteBipartiteGraph(4, 3)
graphs.RandomBipartite(3, 4, p = 1/4)
graphs.RandomTree(36)
```

(d) Some more examples can be found on this Sage documentation page.

Remark 3.1. The vertices of the graphs need not be integers. They could also be strings (entered appropriately with quotes) or tuples of integers, for example. ◁

4 Importing the function

As opposed to the interface given above, it is possible that the reader wishes to import the function `find_order` that takes as input a graph and returns the order as its output. Placing the following snippet of code at the beginning of a `.sage` file (in the same directory as the other files) will do the job.

```
from find_order import find_order
```

The input of `find_order` is a graph object. If the graph is known to be connected, then the reader may wish to use the function `find_order_connected` instead. The following snippet will make it available.

```
from find_order import find_order_connected
```

Both of these functions and their outputs are described in Section 5.

Remark 4.1. The function `find_order_connected` does not check if the input is connected. ◁

If the reader wishes to examine the source code, the relevant file to be viewed is `find_order.sage`. Strictly speaking, the imports happen from the file `find_order.py`. The latter file was generated from the former using the following bash commands.

```
$ sage find_order.sage
$ mv find_order.sage.py find_order.py
```

Part II

Implementation

5 Working of the functions

In Section 6 we will describe the graphs for which the program will give a valid order as an output. Before we do that, we outline the flowchart that the function `find_order` follows. It essentially calls upon another function called `find_order_connected`.

The function `find_order` takes a graph `G` as its input and performs as following.

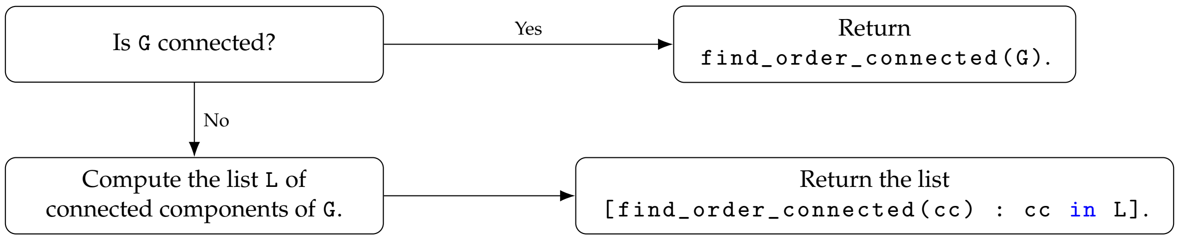
Is G connected?
Yes
Return find_order_connected(G).
No
Compute the list L of connected components of G.
Return the list [find_order_connected(cc) : cc in L].

Figure 1: Description of `find_order`

The function `find_order_connected` takes a connected graph `G` as its input and performs as following.

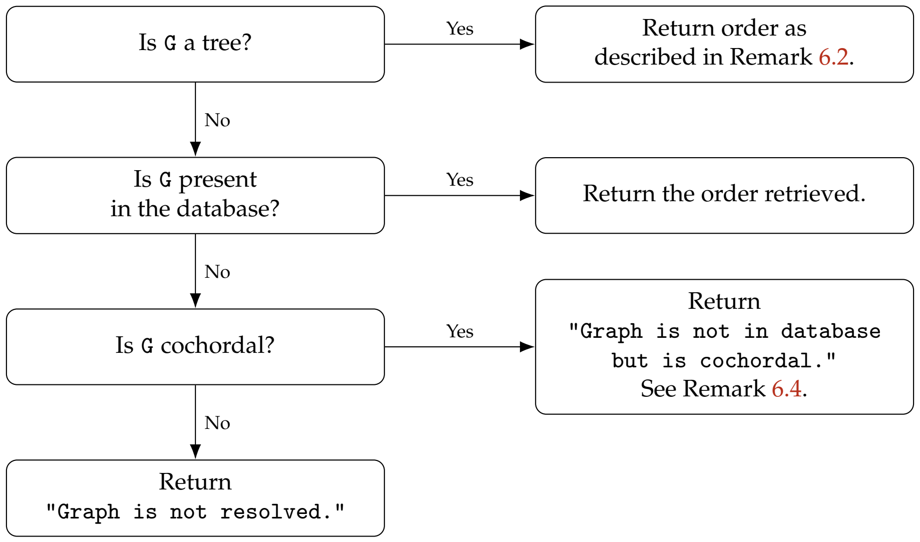
Is G a tree?
Yes
Return order as described in Remark 6.2.
No
Is G present in the database?
Yes
Return the order retrieved.
No
Is G cochordal?
Yes
Return "Graph is not in database but is cochordal." See Remark 6.4.
No
Return "Graph is not resolved."

Figure 2: Description of `find_order_connected`

In the cases that `find_order_connected` does return an order, the object returned is a list of edges of `G`, in ascending order.

6 Graphs for which data is available

In this section, we describe the graphs for which the program will output a valid order. Since we may work with the connected components of the graph, we may as well assume that the input is connected.

6.1 Theoretical results

In this subsection, we recall some graph-theoretical terms and describe how the program deals with certain classes of graphs. We assume that the reader is familiar with the notion of a (simple) graph.

Definition 6.1. For a connected graph G on n vertices, the following conditions are equivalent:

(a) Removing any edge from G makes it non-connected, i.e., G is minimally connected.

(b) G contains no cycle.

(c) G contains $n-1$ edges.

A *tree* is a connected graph satisfying any one of the above conditions.

Remark 6.2. If G is a tree, then it is known that G can be resolved by the GBM algorithm, see [CHM24, Theorem 5.4]. The program implements an algorithm that gives a valid GBM order for any tree on any number of vertices.

The algorithm works as follows: Fix any vertex v of G. To each edge $e = \{u, w\}$ of G, assign the number $d(e) := \min(\text{dist}(v, u), \text{dist}(v, w))$. Then, fix any total order ($\prec$) on the edges that satisfies

$$d(e) > d(e') \Rightarrow e \prec e'.$$

In words: the edges closer to the distinguished vertex are larger in the order.
Any such total order works, and the algorithm will return one such total order. $\triangleleft$

Definition 6.3. A graph is *chordal* if it contains no induced n-cycle for $n \geq 4$. A graph is *cochordal* if its complement is chordal.

Remark 6.4. If G is cochordal, then it known that G can be resolved by a strengthening of the BM algorithm, see [CHM24, Corollary 3.3]. Strictly speaking, this is not the usual GBM, but by abuse of terminology we shall refer to this as GBM as well. $\triangleleft$

6.2 Graphs with at most 7 vertices

Let G be a connected graph with at most 7 vertices.

If G is a tree, then Remark 6.2 applies and we get an order. The database contains orders for almost all non-tree connected graphs that have at most seven vertices. There is exactly one exception—the complete graph on seven vertices. Note however that this is graph is cochordal and thus, Remark 6.4 applies.

To conclude: if G is a connected graph with at most seven vertices, then G is known to be resolved by GBM, and with the exception of K_7, the program will give an explicit order.

6.3 Graphs with 8 or 9 vertices

For connected graphs with 8 vertices, we again have that GBM can resolve all such graphs. For every such non-cochordal graph, the program will give an explicit order. Moreover, the database contains explicit orders for some cochordal graphs as well. For example, if the graph has at most 13 edges, then an explicit order is given. Table 1 gives the exact numbers.

For connected graphs with 9 vertices, we have partial results. It is not known whether every such graph can be minimally resolved by GBM. The database has explicit orders for 212868 such graphs. The database contains no cochordal graphs, in view of Remark 6.4. It contains all non-cochordal connected graphs with

# edges	# connected graphs	# graphs in file	# non-cochordal connected graphs
8	89	89	81
9	236	236	221
10	486	486	453
11	814	814	759
12	1169	1169	1077
13	1454	1454	1325
14	1579	1578	1414
15	1515	1508	1317
16	1290	1270	1076
17	970	922	760
18	658	474	474
19	400	248	248
20	220	113	113
21	114	38	38
22	56	12	12
23	24	3	3
24	11	1	1
$\geq$25	9	0	0
Total	11094	10415	9372

Table 1: Data of connected graphs with 8 vertices.

at most 20 vertices. It contains around 85% of non-cochordal connected graphs with 21 vertices. It contains no graphs with more than 21 vertices. The exact numbers are given in Table 2.

Remark 6.5. If G is a connected graph on n vertices, then G is not a tree iff G has at least n edges. This is why Tables 1 and 2 start with n edges. $\triangleleft$

# edges	# connected graphs	# graphs in file	# non-cochordal connected graphs
9	240	230	230
10	797	777	777
11	2075	2030	2030
12	4495	4409	4409
13	8404	8247	8247
14	13855	13589	13589
15	20303	19898	19898
16	26631	26045	26045
17	31400	30604	30604
18	33366	32365	32365
19	31996	30828	30828
20	27764	26488	26488
21	21817	17358	20501
$\geq$22	37890	0	32621
Total	261033	212868	248632

Table 2: Data of connected graphs with 9 vertices.

7 Retrieval from the database

In this section, we describe how the graphs are stored in the database, and how the data is retrieved. The "database" being referred to in this document is a collection of 31 `.txt` files present in `GBM_order.tar.gz`. Each text file consists of hundreds or thousands of lines. Each line is the list of edges of some graph. The order of the edges in the list is such that it forms a good order. Two different lines in the database always return non-isomorphic graphs. The database contains no tree. In particular, the database contains no graph with one or two vertices.

The graphs are distributed among the different text files based on the number of vertices and edges that they have:

- Graphs with at most 7 vertices are in `3_7_vertices.txt`. The graphs are listed in increasing order of number of vertices. Within the same number of vertices, they are listed in increasing order of number of edges. There are 970 graphs in this file.
- For graphs with 8 or 9 vertices, the graphs are further divided into different files based on the number of edges. There are text files of the form

  `8_vertices_08.txt`, . . . , `8_vertices_24.txt`,   `9_vertices_09.txt`, . . . , `9_vertices_21.txt`.

  If a graph has n vertices and m edges, then it must be present in the file `n_vertices_m.txt` if it is present at all. Within the same file, the graphs are listed in increasing lexicographic order of their degree sequence. In particular, all graphs with one particular degree sequence are listed in a contiguous block of lines within the text file. The file `degree_sequences.txt` contains the interval of these line numbers.

Thus, given a (connected) graph G with at most 9 vertices, the above invariants narrow down the list of lines to check from the database. Within this given list of lines, the program loops through them and creates the corresponding graph H. It then checks whether H and G are isomorphic. If they are, then an isomorphism $H \to G$ is also constructed. Using this isomorphism and the order on the edges of H, the corresponding one on G is returned.
If the loop is over with no isomorphism obtained, then `None` is returned, indicating that the graph is not present in the database. The program may still give some more information as described in Figure 2.

References

[CHM24] Trung Chau, Tài Huy Hà, and Aryaman Maithani. *Monomial ideals with minimal generalized Barile-Macchia resolutions*. 2024. arXiv: 2412.11843 [math.AC].

[Sag20] Sagemath, Inc. *CoCalc – Collaborative Calculation and Data Science*. https://cocalc.com. 2020.

[The23] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 9.8)*. https://www.sagemath.org. 2023.