DOCUMENTATION: FLAGSER-COUNT

JASON P. SMITH

This is an adapted version of Daniel Lütgehetmann's FLAGSER-COUNT (available here) for computing the number of directed cliques in a directed graph, or equivalently the simplex counts of the directed flag complex of the graph. The main differences from the original flagser-count are: all persistent homology code removed to reduce unnecesary memory usage, a compressed option where the graph is stored by flagser-count in sparse format, the ability to read in the graph in sparse format, the functionality to print the simplices in normal form or in a condensed memory efficient form, the functionality to print the number of directed cliques each vertex belongs to, and the ability to select which vertices to consider as source vertices of the clique.

Important: the input graphs can not contain self-loops, i.e. edges that start and end in the same vertex.

1. Requirements

FLAGSER-COUNT requires a C++11 compiler (such as g++) and to install pyflagsercount the pybind11 package is required, available here.

2. Compiling the source code

To compile using g++: Open the command line, change into the FLAGSER main directory and compile with:

```
g++ src/flagser-count.cpp -o flagser-count -std=c++11 -pthread -lz
```

3. Usage

After building FLAGSER-COUNT it can be run as follows:

```
./flagser-count [options]
```

For example:

```
./flagser-count --in-format flagser --in ./test/a.flag --out a.out --threads 8
```

The following `[options]` exist:

- **--in-format *format*:** the input format, can be *flagser*, *edge-list*, *csc* or *coo*
- **--out *filename*:** the simplex counts and Euler characteristic to *filename*, when omitted the results are printed to the terminal
- **--max-dim *dim*:** the maximal dimension to be computed
- **--print *filename*:** prints all the simplices to a text file where each line gives a simplex, a file called filename$i$.simplices is created for $i = 0, ..., number\_of\_threads - 1$. When used with --max-simplices only the maximal simplices are printed.
- **--binary *filename*:** prints all the simplices in a condensed binary format, a file called filename$i$.binary is created for $i = 0, ..., number\_of\_threads - 1$. When used with --max-simplices only the maximal simplices are printed.
- **--max-dim-print *dim*:** the maximum dimension to be printed (inclusive) when –print or –binary are also called
- **--min-dim-print *dim*:** the minimum dimension to be printed (inclusive) when –print or –binary are also called, by default set to 2
- **--size *n*:** the number of vertices in the graph,ımust be given for edge-list, csc and coo formats

- **--threads *t*:** the number of parallel threads to be used, if not specified $t = 8$ is used
- **--containment *filename*:** prints the number of directed cliques each vertex belongs, where line i in *filename* corresponds to vertex i with the number of cliques it belongs to given for each dimension. When used with --max-simplices the number of maximal simplices each vertex belongs to is printed instead. It is recommended to also use –est-max-dim to speed up computation.
- **--edge-containment *filename*:** prints the number of directed cliques each edge belongs, where line i in *filename* is of the form "$i\ j : c_0\ c_1\ \ldots$" where c_k is the number of k simplices which contain edge (i, j). When used with --max-simplices the number of maximal simplices each edge belongs to is printed instead. Requires --edges to be given. It is recommended to also use –est-max-dim to speed up computation.
- **--edges m:** the number of edges in the graph, only needed for edge-containment.
- **--vertices-todo *filename*:** select only certain vertices to be considered as source neurons of cliques, input a 32bit numpy array containing the vertices to be considered.
- **--max-simplices:** also counts the number of maximal simplices, and when used with print, binary or containment it will print the maximal simplices or return the number of maximal simplices each vertex is contained in instead.
- **--est-max-dim:** an estimated value for the maximum dimension, giving a good estimate will make the computations slightly faster (particular when using containment or edge-containment).
- **--compressed:** Enabling this option stores the graph in sparse format, slower performance but significantly less memory required. The graph must be inputted in csr (or csc) format, it is then kept in that format for the computations
- **--transpose:** transposes the graph
- **--progress:** prints the progress of the computation, every time a vertex is finished being considered as a source it prints the current count of that thread and the vertex considered.

4. Formats

4.1. flagser.

The *flagser* format takes as input a flag file by including --in *filename*, which must have the following shape:

```
dim 0:
0 0 ... 0
dim 1:
first_vertex_id_of_edge_0 second_vertex_id_of_edge_0
first_vertex_id_of_edge_1 second_vertex_id_of_edge_1
...
first_vertex_id_of_edge_m second_vertex_id_of_edge_m
```

The edges are oriented to point from the first vertex to the second vertex.

Example. The full directed graph on three vertices is described by the following input file:

```
  dim 0:
  0.2 0.522 4.9
  dim 1:
  0 1
  1 0
  0 2
  2 0
  1 2
  2 1
```

Note the number of vertices on line 1 of the flag file will be the number of vertices used, so --size has no effect when used with flagser format.

4.2. edge-list.

The *edge-list* format takes as input a text file by including --in *filename*, where every line of the file has two integers separated by whitespace which represents an edge from the first integer to the second. This is equivalent to flagser format with the first three lines removed.

4.3. csc.

Uses scipy csc format, also requires

```
--indices filename1 --indptr filename2
```

where the two files are numpy arrays corresponding to the indices and indptr lists in scipy csc format.

4.4. csr.

Uses scipy csr format, same input as csc.

4.5. coo.

Uses scipy coo format, also requires

```
--row filename1 --col filename2
```

where the two files are numpy arrays corresponding to the row and column lists in scipy coo format, i.e two lists were (row[i],column[i]) is an edge of the graph, for all i.

All inputted numpy arrays should be either uint32 or uint64. Flagser-count will automatically detect which is used. Inputting int32 or int64 is also acceptable. Any other inputted type will throw an error.

5. Binary output

When using the --binary the simplices are printed in the following way: Each simplex is represented as a sequence of 64 bit integers. With each 64 bit integer representing 3 vertex id's stored in 21bit format. The simplices are ordered from right to left. If the leading bit is 0 this indicates the start of a new simplex. So if b is the binary representation of a 64 bit int, then $b[0]$ indicates whether we start a new simplex, $b[43:64]$, $b[22:43]$ and $b[1:22]$ are the binary representations of the vertex id of the simplex, in that order. So in a 2 dimensional simplex $b[43:64]$ is the source and $b[1:22]$ is the sink and $b[0]=0$. See the included python script binary2simplex.py for code to extract the simplices. The number of vertices must be less than 2^{21}

The size (in bytes) required to print the simplices can be computed, when the simplex counts are known, using the following formula:

$$\sum_{i=2}^{d} C_i \left\lceil \frac{i}{3} \right\rceil 8$$

where C_i is the number of simplices in dimension i. Assuming we don't print the vertices and edges, which is the default as these are already known.

6. progress

The --progress flag returns the counts computed on each thread so far after consider each vertex. For example calling

```
./flagser-count --in-format coo --row ./test/C_row.npy --col ./test/C_col.npy --size 5
                --threads 2 --progress
```

Gives the output:

```
1 : thread 1 : 1 1  : 1/5
3 : thread 1 : 2 4 3 1  : 2/5
0 : thread 0 : 1 2 1  : 3/5
2 : thread 0 : 2 2 1  : 4/5
4 : thread 0 : 3 5 4 1  : 5/5
```

where the first element is the gid of the vertex which has just finished computing, the second entry is the thread it was computed on, the third entry is the current count on that thread, and the final entry is the number of neighbours that have finished computing.

So to get the counts for which vertex 3 is the source you take the third entry of the line '3 : thread 1 : 2 4 3 1 : 2/5', which is 2 4 3 1, and subtract from it the counts for the last time before that thread 1 appears which is the line '1 : thread 1 : 1 1 : 1/5', thus the counts for vertex 3 as source are 1 3 3 1.

7. PYFLAGSERCOUNT

pyflagsercount is a python wrapper (using pybind11) for flagser-count. To use *pyflagsercount*, first ensure it is installed, then open python and load the package with:

```
from pyflagsercount import flagser_count
```

Then simply call:

```
X=flagser_count(M)
```

Where M is the adjacency matrix of the graph to consider as a numpy array.

The function flagser_count returns a dictionary containing the Euler characteristic (X['euler']) and the cell counts (X['cell_counts']), along with the maximal cell counts (X['max_cell_counts']) and containment counts (X['contain_counts']), when the max_simplices or containment option are called, respectively.

All of the optional arguments can also be used with *pyflagsercount* (except --progress), by using them as input arguments in flagser_count, for example to specify 10 threads call:

```
X=flagser_count(M,threads=10)
```

Or to print the simplices to outfile call:

```
X=flagser_count(M,print='outfile')
```

To use compressed version use compressed=True. To return containment values use containment=True, the containment values will be returned in the dictionary and will not be printed to a file.

The option return_simplices=True, is available and when used will return the simplices in the dictionary so can be accessed with X['simplices']. The output is separated by dimension, so X['simplices'][3] will be all simplices of dimension 3. Note that dimensions 0 and 1 are always empty as the vertices and edges are already known.

The functions *flagser_count_edges* and *flagser_count_csr* are also available, the optional arguments for these are the same as flagser_count. The required arguments are:

```
X=flagser_count_edges(number_of_vertices, list_of_edges)
```

where the list of edges is a list of pairs with each pair representing an edge, and

```
X=flagser_count_csr(number_of_vertices, indices_address, indptr_address)
```

where indices_address is a string containing the destination of a numpy file which contains the indices of the matrix when stored as csr format, similarly for indptr_address.

8. FLAGSER-COUNT-INDIVID

A version of flagser that has a required input --vertex v, and computes all simplices in the graph for which v is the source. Will return the same value as calling flagser-count where vertices_todo contains the single vertex v. However, this version will consider each neighbour of v on a separate thread, whereas using flagser-count will use a single thread for the whole computation. This is useful when considering vertices with very high out degree. Accepts all the same flags as flagser-count listed above.

The --progress flag will return something similar as that for flagser-count, except the first entry is the neighbour of v that has finished computing.

For example (without the linebreak):

```
./flagser-count-individ --in-format coo --row ./test/C_row.npy --col ./test/C_col.npy
--size 5 --threads 2 --vertex 3 --progress
```

University of Aberdeen, Aberdeen, United Kingdom
Email address: jason.smith@abdn.ac.uk