DOCUMENTATION: FLAGSER

DANIEL LÜTGEHETMANN

The software FLAGSER is an adaptation of Ulrich Bauer's RIPSER, enabling the computation of persistent homology for directed flag complexes with different filtrations.

1. Requirements

FLAGSER requires a C++14 compiler and CMake. For custom filtration algorithms, additionally PYTHON (including PIP) are required. To see the PYTHON packages needed in this case, see `requirements.txt`. If you want to read .h5 files, you need to install the HDF5 library as well (https://support.hdfgroup.org/HDF5/).

2. Compiling the source code

Open the command line, change into the FLAGSER main directoy and execute

```
(mkdir -p build && cd build && cmake .. && make -j)
```

Specific for MacOS: on the first run, you may be asked to install the XCode Developer Tools, which you should confirm. After the automatic installation finished, just run the command above again. For the preinstalled PYTHON there might be problems installing the necessary libraries. If the compilation is not successful, execute `sudo easy_install pip` and try again.

3. Usage

After building FLAGSER, you can compute persistent homology as follows:

```
./flagser [options] filename
```

You can slightly increase the speed of computation by using more memory:

```
./flagser-memory [options] filename
```

The following `[options]` exist:

- **--out *filename*:** write the barcodes to the given file
- **--out-format *format*:** the output format, which is either `barcodes` or `betti`. The default is `barcodes`
- **--in-format *format*:** the input format, which is either `h5` for HDF5 files or `flagser` for a FLAGSER file (see below). For the h5 input format the HDF5 library needs to be installed before building the source code. The format defaults to `flagser`

--h5-type *type*: the type of data in the h5-file. The type can either be `"matrix"` if at the given path in the HDF5-file there is the connectivity matrix or `"grouped"` if the connectivity matrices are grouped. To only consider a subset of the groups you can list them after `"grouped"`, e.g. `"grouped:L1_DAC,L2*"`. The star is a placeholder for arbitrary characters. The type defaults to `"matrix"` and is only relevant for the input type `h5`

--filtration *algorithm*: use the specified algorithm to compute the filtration. **Warning**: if an edge filtration is specified, it is assumed that the resulting filtration is consistent, meaning that the filtration value of every simplex of dimension at least two should evaluate to a value that is at least the maximal value of the filtration values of its containing edges. For performance reasons, this is not checked automatically.

--max-dim *dim*: the maximal homology dimension to be computed

--min-dim *dim*: the minimal homology dimension to be computed

--approximate *n*: skip all cells creating columns in the reduction matrix with more than n entries. Use this for hard problems, a good value is often 100000. Increase for higher precision, decrease for faster computation

--components: compute the directed flag complex for each individual connected component of the input graph. *Warning: this currently only works for the trivial filtration. Additionally, this ignores all isolated vertices.*

--help: print a help screen

The available filtration algorithms are printed when executing `./flagser --help`. For the input format `h5`, you can select the path inside the h5-file by appending it to the filename, e.g. `./flagser ...filename.h5/experiment1/connectivity`.

The input file defines the directed graph and must have the following shape (if the option `--h5` is not used):

```
dim 0:
weight_vertex_0 weight_vertex_1 ...  weight_vertex_n
dim 1:
first_vertex_id_of_edge_0 second_vertex_id_of_edge_0 [weight_edge_0]
first_vertex_id_of_edge_1 second_vertex_id_of_edge_1 [weight_edge_1]
...
first_vertex_id_of_edge_m second_vertex_id_of_edge_m [weight_edge_m]
```

The edges are oriented to point from the first vertex to the second vertex, the weights of the edges are optional. The weights should be rational numbers.

Example. The full directed graph on three vertices without explicit edge weights 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
```

4. Tools

You can use the following tools to generate input matrices for FLAGSER.

4.1. **H5 to flagser.** To convert a connection matrix in the H5-format to an input file for FLAGSER, call

```
./tools/h5toflagser [--groups "GROUP1,GROUP2,..."]
    [--random-edge-filtration] h5file output_filename
```

With the parameter `groups` you can extract the subgraph consisting only of neurons in the given groups, otherwise the full graph is constructed. You can use a star in the group names to match multiple groups, e.g. `--groups "L5_*"` matches all groups in layer 5. If you pass the parameter `--random-edge-filtration`, then each edge is assigned a random weight in $[0, 1]$.

Example. To construct the full second column of the average rat, run

```
./tools/h5toflagser cons_locs_pathways_mc2_Column.h5
      cons_locs_pathways_mc2_Column.flag
```

To only extract layer 4 and layer 5, run

```
./tools/h5toflagser --groups "L4_*,L5_*"
      cons_locs_pathways_mc2_Column.h5
      cons_locs_pathways_mc2_Column.flag
```

4.2. **Erdős–Rényi graphs.** To construct an Erdős–Rényi graph on n vertices with density $d \in [0, 1]$, call

```
./tools/er [--density d] [--random-edge-filtration] n output_filename
```

If the density is not specified, it is assumed to be $d = 1$. If you pass the parameter `--random-edge-filtration`, then each edge is assigned a random weight in $[0, 1]$.

Example. Calling `./tools/er --density 0.314 20 er-20.flag` creates the FLAGSER input file for a graph on 20 vertices where 31.4% of all *directed* edges are present.

4.3. **Counting flag complex cells.** To just count the number of cells of the directed flag complex, call

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

You can use the same input options as for `flagser`. If you specify an output file by

```
./flagser-count --out filename.h5 ...
```

then a list of cells is saved into an existing or new HDF5-file. If you specify `--out filename.h5/some/path`, then the data will be written into the subgroup `/some/path`.

5. Writing custom filtration algorithms

The filtration of the directed flag complex is computed by algorithms. Examples of such algorithms are `max` (assigning each cell the maximal weight of its boundary cells) and zero (assigning each cell the weight zero). To implement a custom algorithm you extend the file `algorithms.math` by another definition. This definition needs to be written in a special tiny programming language resembling mathematical notation. It is described in detail below, but here is an example:

Example. The code

```
square_mean = max(faceWeights)
  + sum(faceWeights, x -> x^2)^(1/2) / (dimension + 1)
```

assigns each cell σ the weight

$$f(\sigma) = \max_{\nu \in \partial(\sigma)} f(\nu) + \left(\Sigma_{\nu \in \partial(\sigma)} f(\nu)^2\right)^{1/2} \cdot \frac{1}{\dim(\sigma) + 1}$$

You can find more examples by looking at the built-in functions in `algorithms.math`.

5.1. **Defining a filtration algorithm.** To add a new filtration algorithm, think of a name (consisting only of the letters a to z and underscores) and add a new line of the form

```
your_chosen_name [overrideVertices] [overrideEdges]
    = formula_to_compute_the_weight
```

into `algorithms.math`, where `formula_to_compute_the_weight` is an expression as described in the next section. The optional keywords `overrideVertices` and `overrideEdges` indicate that the weights given to the vertices and edges in the input file are replaced by values computed via this algorithm. For example, the trivial filtration algorithm reads as follows:

```
zero overrideVertices overrideEdges = 0
```

You have to recompile FLAGSER by executing `make` again and can then use the new filtration algorithm by passing the option `--filtration your_chosen_name` to FLAGSER.

5.2. **The mathematical input language.** The language for defining filtration algorithms is given in a special format, described in this section. You can use basic arithmetic expressions `+, -, *, /, ^` and the exponential map `exp( )`. In addition to that, you can access several constants like the dimension of the current cell or weights, as well as built-in functions. Below we give a list of all these available constants and commands.

On the top level you can perform a case distinction by dimension:
cases: dim *n1*: *formula1* dim *n2*: *formula2* ... else: *formula*

Example. To assign vertices and edges the value 0 and all other faces their dimension as weights, we would define

```
modified_dimension = cases:
  dim 0: 0
  dim 1: 0
  else: dimension
```

If an algorithm requires a certain filtration to be provided by the input file (which is only reasonable for dimension 0 and 1 of course), then you can add an error message shown to the user whenever this data is missing by using **error *error_message***

Example. If you require weights for all vertices and edges, you write a filtration algorithm as follows:

```
needs_vertex_and_edge_weights = cases:
  dim 0: error "Please provide weights for the vertices."
```

```
dim 1: error "Please provide weights for the edges."
else: some_formula
```

As available constants for your formulas you can use the following:

dimension: the dimension of the current cell
faceWeights: the list of weights of the faces of the current cell
cellVertices: the ordered list of vertices of the current cell
vertexOutDegrees: the outgoing degrees of all vertices of the graph
vertexInDegrees: the incoming degrees of all vertices of the graph
vertexWeights: the weights of all vertices of the graph

Example. You get individual entries of the lists above by appending `[index]` to them, e.g.

```
cellVertices[0]
```

gives the first vertex in the cell,

```
faceWeights[dimension]
```

gives the weight of the last face of the current cell and

```
vertexInDegrees[cellVertices[2]]
```

gives the incoming degree of the third (notice the indexing starting from zero) vertex of the current cell.

For the lists (except for the vertex degrees and the vertex weight) you can use the following functions:
min/max(*list* [, *from*, *to*]): These commands return the minimal and maximal element of the given list. If you pass `from` and `to`, the elements before `from` and after `to` are disregarded. If the specified part of the list is empty, then 0 is returned.

Example. The following examples show the usage of those two functions.

```
maximum = max(faceWeights)
maximum_plus_minimum_of_first_three_faces
    = max(faceWeights) + min(faceWeights, 0, 2)
```

sum/product(*list* [, *from*, *to*] [, *function*]): These commands return the sum or the product of the elements in the given list. If you pass `from` and `to`, the elements before `from` and after `to` are disregarded. If the specified part of the list is empty, then 0 is returned.

If you pass a `function`, each element is mapped through this function before adding or multiplying. A **function** is defined in the form `x -> expression_in_x`, for example

```
x -> x^2
```

denotes the function that squares each element,

```
z -> 3*z-5*exp(1/z)
```

denotes the function $z \mapsto 3z - 5e^{1/z}$.

Example. The following examples show the usage of those two functions.

```
sum = sum(faceWeights)
sum_squared = sum(faceWeights, 2, dimension, x -> x^2)
```

```
product_vertex_weights
    = product(cellVertices, v -> vertexWeights[v])
```

combine/reduce(*list* [, *from*, *to*], *function*, *initial_value*): These two commands are the same, and they combine elements in a list into a single number. If you pass `from` and `to`, the elements before `from` and after `to` are disregarded. If the specified part of the list is empty, then `initial_value` is returned.

The `function` you pass must take two elements and produce a third one, i.e. it must be of the form `carry x -> expression_in_carry_and_x`. The first argument of the function is the accumulated result, the second one is the next value. In the first step, `initial_value` is passed as the accumulated result.

For example, we have the following equivalences:

```
sum(array) == combine(array, carry x -> carry + x, 0)
product(array) == combine(array, carry x -> carry * x, 1)
```

Example. The following example shows the usage of this function.

```
squaring = combine(
  faceWeights,
  carry x -> (carry + x)^2,
  vertexWeights[cellVertices[0]]
)
```

University of Aberdeen, Aberdeen, United Kingdom
Email address: daniel.lutgehetmann@abdn.ac.uk
URL: https://homepages.abdn.ac.uk/daniel.lutgehetmann/pages/