PROJET TDA

26 October 2017

Aurélien Chaline, Maxime Bourliatoux

1 STRUCTURE OF THE CODE

Our code is composed of one main class, *ReadFiltration*, which reads the input given and put it into an easy to manipulate structure. Then it calls the functions from the computing class.

There are two computing classes. The first one using lists of lists to represent matrices, and the second one a sparse representation. These classes implement the creation of the matrix from the filtration, the reduction algorithm, the creation of the barcode and the printing of the matrix.

Finally we have a class for the sparse matrix representation.

1.1 Computing with usual matrix implementation

The first class called *Compute*, uses lists of lists to represent matrices. The implementations of the algorithms are not optimised and very slow, using only iterations over each indices of the lists storing the values, which is not really useful because there is many zeros. The high number of zeros, combined to the huge quantity of data quickly leads to *Out of Memory* errors, while only a few indices are really useful. That's why we switched to sparse matrices representation.

1.2 Sparse Matrix implementation

As our matrices are made of zeros and ones, we can use a special representation that does not store the value of the element. We choose to use HashMaps, storing lists containing the indices where they were '1's. We made a double representation using Maps for the columns with lists for the row, but also the opposite. This uses more memory but allows easier and faster manipulations.

Then we created the useful functions, for adding or removing an element, adding columns together but also more specific functions to ease the reduction. Our adding functions finds the list associated to the column in O(1) with the hashmap, and the adding of the element at the end of the list is linear. The same thing occurs for removing and finding.

1.3 Computing with sparse matrices

The idea for computing is close to the one used with usual matrices. Still we tried to make this more efficient and less costly. We tried to use HashMaps to store the needed data and access them in O(1), instead of iterate over a whole line to find the elements, but adding the data to the maps was also costly.

2 REDUCTION

Q3 For the normal implementation, our reduction code is composed of a for loop nested in a while loop nested in a for loop. Each of the loops occurring m times. So the complexity of the reduction is about $O(m^3)$.

3 EXPERIMENTS

3.1 Classical spaces

Q5 Our filtrations for the Moebius band, the Torus, the Klein bottle and the projective plane are shown on the Figure below.

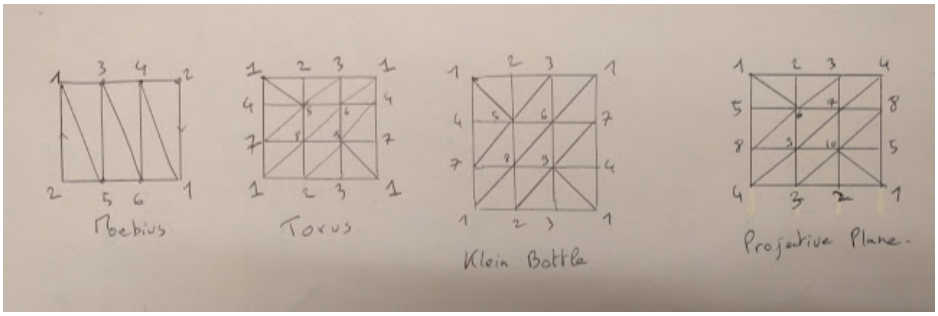

Figure 1 – Filtrations' drawings

For the n-ball and n-sphere, we can make a recursive algorithm. Indeed, when you have a ball in one dimension put into triangles, you just need to add a new point, and link it with all the existing point, creating the balls in all lower dimensions, and finally reaching a new dimension.

Q6 The barcodes are in the files named barcode{Moebius, Torus, Klein, Plane}.txt

3.2 Simulated data

Q7 We only succeeded in computing filtration B. It took us nearly 2 hours. We stopped filtrations A and C after more than 3 hours running. Filtration D caused a memory error that we haven't been able to solve.

Q8

— **A :** There were a lot of dimension 0 components but only one component is left after the value -13. We can see that H1 is increasing then every interval stop between -13 and -12. There one important bar in H_3 from -10 to -6.5.
— **B :** We can see from 0 to 0.7, a space with $H_0 = 1$, $H_1 = 5$ and $H_2 = 8$. Then there is a zone where we only have $H_0 = 1$ and $H_2 = 1$, from 1.2 to 3.
— **C :** We can see that if there were a lot of components in dimensions 0 and 1 in 0, there is from 0.01 to 1 $H_0 = 1$, $H_1 = 2$ and $H_2 = 1$. Then from 1 to 9 there is one connected component and one hole in H_1.
— **D :** We can notice the interval [-5, 0] where we have $H_0 = 1$, $H_1 = 2$ and $H_2 = 1$