Efficient implementations of the stochastic block model

Luca Gandolfi

1 Introduction

Last year, as a part of my Bachelor Thesis I wrote in Python three implementations of three different Stochastic Block Model (SBM) algorithms used to perform community detection on graphs. The goal of the implementation part of my work was to understand the algorithms and to have a working implementation of them, thus I didn't focus on their efficiency. Consequently, the algorithms written in Python were quite slow and thus difficult to use on large graphs. Moreover, for all of the three variants performances were also crucial in order to obtain a reliable result, indeed two out of three variants used a Monte Carlo Markov Chain approach, and thus being able to perform more iterations of the Markov chain implies being more able to reach convergence. The last one instead, was a greedy algorithm that was repeated with different random initializations, keeping the best result at the end; thus being able to run the algorithm with more random initializations implies having better results. The goal of this project is to implement them (two of them) efficiently in C++, in order to increase their speed; moreover, I haven't found C/C++ implementations of them online, therefore they could be useful also to other people. Note however that my focus has been purely on the algorithms, so a little more work should be done to use them in practice, mainly for taking in input the adjacency matrix.

Note, throughout this report I will test the algorithms on a very simple random graph model: given n the number of nodes, I fix $k = 4$ be the number of classes, each of them having the same prior probability, and I let the probability of having an edge between node of different classes equal to γ, while equal to 4γ between nodes of the same class; γ is obtained after fixing a desired expected degree (equal for each node), which I'll call E. For the test, I'll use an i5-9300H CPU, which has 4 cores and 8 threads.

2 Degree corrected SBM

2.1 The starting point and a first implementation in c++

I will start with the Degree Corrected SBM algorithm, since it assumes a more general structure on the graph in input, and thus it generally works better with graphs describing real networks. The algorithm aims to find the assignment that maximizes the loglikelihood under the assumption of the degree corrected SBM, which, differently from the standard SBM allows to have nodes in the same class with different expected degrees. To do so it runs a greedy algorithm that, starting from a random initializations, tries to modify it at each iteration until it cannot move anymore. To have better results, it is better to run the algorithm more times, with different random initializations, and keep the result with the highest loglikelihood. In the following analysis, I will run the algorithm on 8 different random initializations .

With $n = 100$ and $E = 34$, the original python implementation takes 90.1 seconds[1].

As a first (non-trivial) step I implemented the code in C++ in the exact same way I did in Python. Compiling it with g++ with optimization -O3 and with -march=native, with $n = 100$ and $E = 34$ the algorithm takes 0,259 seconds, already an enormous improvement of almost 350x. However, increasing the dimension of the graph, the time needed increases a lot: with $E = 70$ it takes 5,54s with $n = 300$ and 36,2s with $n = 500$. This is version v1 on the table in 1a.

2.2 First improvements: update() and change_in_loglikel()

After running cprof (and perf), it is possible to see that, with $n = 300$, around 93% of the time spent by the algorithm is on the function *GreedyAlgorithm.update()*, which is used to update the matrices m,

[1] In this analysis the results shown are the time of a single inference, thus their main purpose is to show their order of magnitude, rather than exact times

kit and the vector *kappa* when a new configuration is tried. Thus I started by trying to optimize this function. The relevant observation is the following: most of the times the function is called when the new class assignment is equal to the original one except for one single node: thus, it is useless to recompute m, *kit* and *kappa* from zero, but we can just update them starting from the previous ones. For this reason I created the function *GreedyAlgorithm.fast_update(node, new_class, old_class)*, used to deal with a swap of a single node. This is version v2 on 1a

Now, with $n = 300$, the largest amount of time is spent in the *GreedyAlgorithm.change_in_loglikel(node, old_class, new_class)*, around 83% of the time. The formula to compute the change in loglikelihood is at the heart of the algorithm: we cannot change it but we can try to compute it more efficiently. The first observation is that the *change_in_loglikel()* function is computed a very large amount of times, $O(n)$ times more than any other function. The second observation is: the slowest computation is calculating the function $b(x) = x \log(x)$, with x integer. Now, for any graph, the total number of edges plus 2 times the maximum degree among the nodes is an upper bound of x, so we can pre-compute all the possible values of $b(x)$ and retrieve them each time. This is version v3 in 1a.
At this point it takes 0,634s with $n = 500$, but still 27,4s with $n = 2000$.

2.3 Parallelization, change of types, minor changes

Clearly the algorithm is easily parallelizable, simply by executing in parallel the random initializations. I did it with openMP, creating an instance of the class for every random initializations, so that each sub-process has its own variables. This is Version v5 on 1a.

Up to this point, the adjacency matrix Y and other matrices and vectors were instances of Eigen matrices or vectors. However I noticed that accessing their elements requires more time, so I changed everything in std::vector, making the matrices flat vectors. In particular Y became a vector of type *char* (before had type *int*), which allowed to reduce the dimension of Y and to diminish the cache misses with large n. Moreover I did some other minor updates in the code. This is version v10 on 1a.
Now it takes 2,72s with $n = 2000$ and 48,5s with $n = 7500$.

2.4 Now?

At this point, the function *change_in_loglikel()* takes the majority of the time. I've tried many things, but I haven't found any way to improve it. Moreover, the time taken (with $k = 4$) is equally split between the first part, the one with the for loop, and the second part, the one with the five "+=" operations, so the for loop and the if statement are not the problem. Also, according to vTune, only a very small amount of time is taken by the "[]" operator. This function is called n times more times than any other, so it makes sense that it is the bottleneck.

3 Mixture of Finite Mixtures SBM

3.1 Starting point and first implementation in C++

The main characteristic of the MFM SBM algorithm is that it doesn't assume a known number of classes k, so the inference is made not only for the class assignment, but also for the number of classes. It follows a Monte Carlo Markov Chain approach, using a Gibbs sampler to estimate the quantities of interest (the probability matrix of edges W and the class assignment X). The underlying model is a Mixture of Finite Mixtures (MFM), which is then written as a "Chinese Restaurant Process" (CRP) in order to obtain the sampler.

As it is an MCMC method, we need to specify the number of iterations we do: throughout the whole analysis I will keep the number of iterations to 1000. For $n = 100$, $E = 34$ the Python implementation takes 86,2 seconds.

As a first step, as before, I translated in C++ the algorithm, in the exact same way as it was written in Python. Now, with the same parameters, it takes 1,21s. However, as before, increasing n, the amount of time needed increases a lot: with $E = 70$ it takes 9,27s for $n = 300$ and 25,1s for $n = 500$. This is version v1 on 1b.

3.2 A first improvement: log_proba_existing_table()

Using vTune it is possible to see that most of the time was spent in computing the logarithm in the function *CollapsedGibbsSampler.log_proba_existing_table()*[2]. One of the approaches I tried has been to try to compute this logarithm faster, using either the tag -ffast-math or the Intel compiler instead of g++, which claims to provide much faster computations for specific functions as the logarithm, but that wasn't the point. The first big improvement follows from the observation that the computation of the logarithm could have been grouped at two different levels and therefore computing the logarithm that many times was useless as the computations could have been reused. So I changed the structure of that part, obtaining that the logarithm was computed only once at each iteration (excluding when a new class is added) entry-wise on two matrices (W and 1 - W), and that the wanted result for each node is computed by a vector matrix multiplication of the matrices of the logarithms and two vectors which I called *edges* and *non_edges*.
This is version v3 on 1b. At this point, for $n = 500$ it takes 2,7s. While it starts struggling with $n = 2000$, where it takes 53,5s.

3.3 Many more improvements

From now on the way I worked had been the same: I used Vtune to see which function took the most time and then I tried to optimize it, either by pre-computing the values of some specific mathematical functions at the beginning of the algorithm (logarithms at fixed integer steps, or logarithm of the gamma function at various fixed steps), or by changing the way I computed the various parameters, in order to reduce the computations as much as possible: for instance with the creation of the matrices *Abar0, Abar1, matrix_edges, matrix_non_edges* and changing the functions *m()*, *Vn()*, and the old *Abar()*, substituted by the two matrices and by the functions *update_Abar()* and *fast_update_Abar()*. This is version v8 on 1b. Now it takes 0,757 with $n = 2000$, 6,33s with $n = 7500$ and 132s with $n = 18000$.[3]

3.4 Now?

At this point, most of the time is taken inside the step2() function. Using vTune, With $n = 10000$, a third of it is taken by calling the function *update_edges_i()* and another smaller part by sampling the new class.

4 Note on the code

There are two C++ files for the DC SBM. The one that has to be compiled is DC_SBM.cpp, which is used to build the graph and to call the algorithm, while the other, GreedyAlgo.cpp contains the code for the algorithm. The structure is similar also for the MFM SBM (plus there is the file called Beta.cpp that contains some functions used).
They can be compiled as follows:
g++ -O3 -march=native -o [exe_name_DC] DC_SBM.cpp
After, it is possible to test them with:
./exe_name_DC [number of nodes] [Expected degree] [number of initializations]
./exe_name_MFM [number of nodes] [Expected degree] [number of iterations]
As said before, the algorithms in *GreedyAlgo.cpp* and in *CollapsedGibbs.cpp* are fully general, however in this analysis are only tested with the simple model described before, created by the class *StochasticBlockModel*. Note that the DC SBM allows for multiple edges and self-edges, while the MFM SBM doesn't allow for them. The NMI (Normalize Mutual Information) score is a metric of their accuracy, with respect to the true class assignment. NMI = 1 means perfect recovery, and almost all the tests tried achieved it (since the model tested is quite simple).

[2] All the probabilities are computer under the logarithm, in order to numeric explosion
[3] Note however that starting with $n = 16000$ it would be better to do more than 1000 iterations.

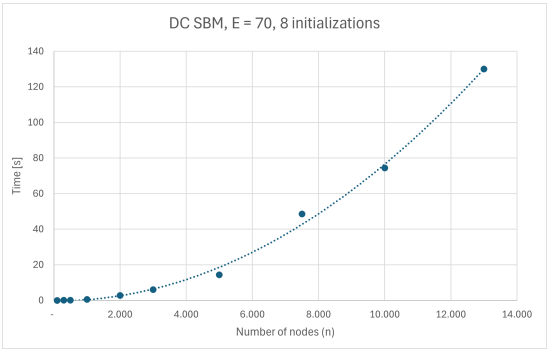
DC SBM, 8 Initializations							
E	n	Python [s]	v1 [s]	v2 [s]	v3[s]	v5 [s]	v10 [s]
34	100	90,1	0,259	0,095	0,020	0,011	0,011
70	300		5,54	1,12	0,215	0,057	0,060
70	500		36,2	3,84	0,634	0,191	0,109
70	1.000			17,8	3,26	1,22	0,502
70	2.000			76,6	27,4	7,73	2,72
70	3.000					24,4	6,01
70	5.000						14,3
70	7.500						48,5
70	10.000						74,4
70	13.000						130

(a) DC SBM.

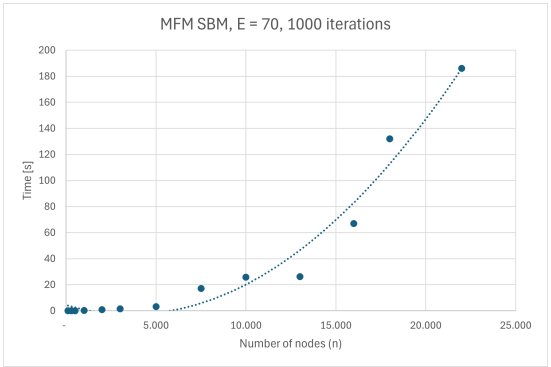
MFM SBM, 1000 iterations					
E	n	Python [s]	v1 [s]	v3 [s]	v8 [s]
34	100	86,2	1,21	0,232	0,028
70	300		9,27	1,14	0,082
70	500		25,1	2,70	0,143
70	1.000			12,7	0,322
70	2.000			53,5	0,832
70	3.000				1,64
70	5.000				3,22
70	7.500				17,1
70	10.000				25,8
70	13.000				26,3
70	16.000				66,9
70	18.000				132
70	22.000				186

(b) MFM SBM.

Figure 1: Time taken by the different versions of the code. Single trial.



(a) v8 DC SBM. Quadratic fit.



(b) v10 MFM SBM. Quadratic fit.

Figure 2: Quadratic dependence of the last version of the algorithms