Experiments With Connected Components, ER Graphs and DBSCAN

Roger Fabien Nabbout Charles

May 2022

All the code and the experimental results can be found on
`https://github.com/ejlly/INF442`.

1 The Tarjan Algorithm

The assignment was to find and implement a sequential algorithm that computes the strongly connected components of a digraph.

Here, we took the Tarjan algorithm[1] and implemented it in C++. Because we need to keep certain values in memory for each and every node in the graph, we created a structure as not to have global variables left in our program.

That is why a `struct Context` can be seen in the `tarjan.hpp` file. We have to keep at all time : the current index `index` of the node we next node to visit, a stack `pile` as to know which nodes to visit next, the order of all points in the `int[]` named `order[]` (order in which points are being discovered), another index for all points in the `int[]` named `dfs_seen`, a `bool[]` named `on_stack` which describes if a point is already in the stack, and ready to be processed, the components strongly connected : `connected_components` and obviously the graph itself.

The time complexity of this algorithm is optimal, as explained in the subject. Indeed, we notice that each vertex is explored at most twice, once when we explore it, and each exploration costs $O(1)$ operations, and once more when we add the related node to the current connected component being filled (when the stack is being emptied in the `check_composante` function. Thus, this algorithm is indeed linear : $O(|E| + |V|)$

Running this algorithm on the *Social circles* dataset from Facebook[3] (duplicating each edge, given that this is an undirected graph), we can see that there is only one connected component.

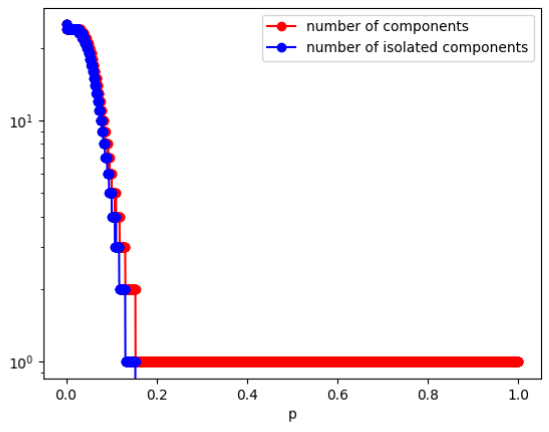
2 Erdös-Rényi Graphs

2.1 Generating ER graphs

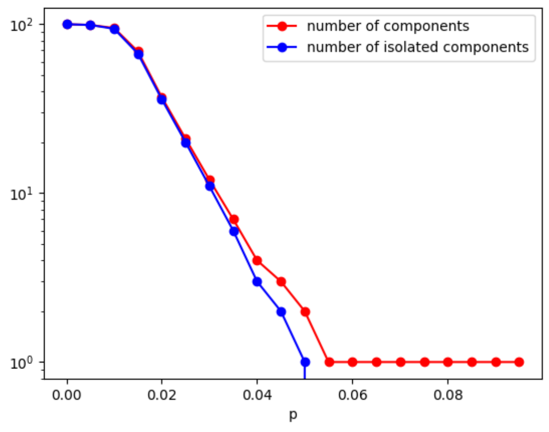
We first generate Erdös-Rényi graphs by iterating over pairs of nodes, and choosing at random if there is an edge between them or not: there is one with probability p. This runs in $O(|E|^2)$. It could have been improved to $O(|V| + |E|)$ (which is much better for very small values of p) by drawing out at random the number X of edges coming out of a given node, where X is drawn from a binomial distribution $B(n, p)$, and then choosing at random which are the outgoing edges. This can be done in $O(1 + np)$ by using the inverse transform method[2], resulting in a total runtime of $O(|V| + |E|)$. However, given the increased complexity of the method, and the fact that it was not necessary to test very large ER graphs to get interesting results, we chose to stick to the simpler $O(|E|^2)$ algorithm.

2.2 Properties of ER graphs

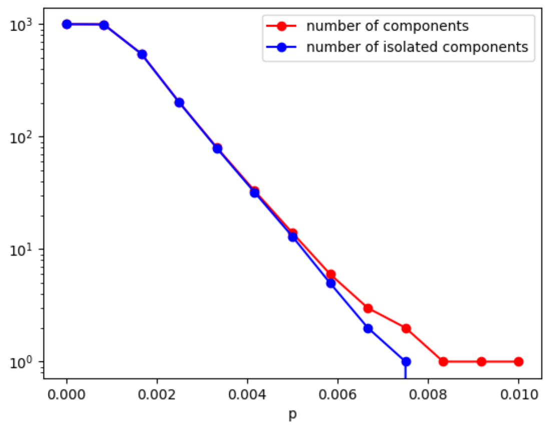
We then use our tarjan function to do measure when ER graphs become fully connected. 1000 ER graphs are generated for each of the values of (n, p) tested. Results are shown in Figure 1.



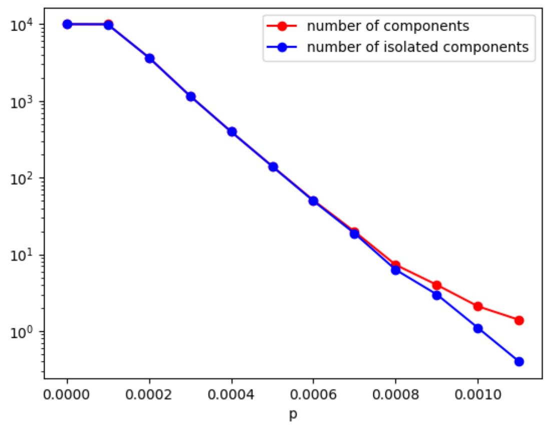
(a) 25 points



(b) 100 points



(c) 1000 points



(d) 10000 points

Figure 1: number of connected component vs p in ER Graphs

We notice that there is a relatively sharp drop when plotting the number of connected components, C, and the number of isolated components, IC, against p (at a given n) at a certain value $t(n)$ where suddenly there is only one connected component left. This seems to be analogous to the phenomenon of criticality seen in the INF371 course.

This drop seems to happen in three consecutive phases, no matter the value of n. (The threshold $t(n)$ between the different phases have not been measured precisely, and are given only to give a rough idea of the phenomenon observed). First, when $p \leq 0.2t(n)$, non-isolated components have a very low likelihood of existing. When $0.2t(n) \leq p \leq 0.6t(n)$, $C/IC \approx 1$, the fraction of non-isolated components is still negligible, but because an increasing number of nodes are becoming connected, the number of isolated components starts to drop (it seems to be that during this step, $\log(C) \approx -ap + p_0$ where $a > 0$). Then, when $0.6t(n) \leq p \leq 0.95t(n)$, none-isolated components become none-negligible. When $0.95t(n) \leq p \leq t(n)$, there are only a couple of connected components left, with no isolated components (with a high probability). Finally, when $p \geq t(n)$, there is only one connected component left.

We didn't study the function $t(n)$ in details, but it seems that $t(n) \approx \frac{1}{n}$. For <u>undirected</u> ER graphs, it is known that the threshold is of order $\frac{\log(n)}{n}$[4]. Knowing it is much easier for an undirected graph to be connected than for a directed one, what we measured seems to agree with the theory.

3 DBSCAN

3.1 The DBSCAN algorithm

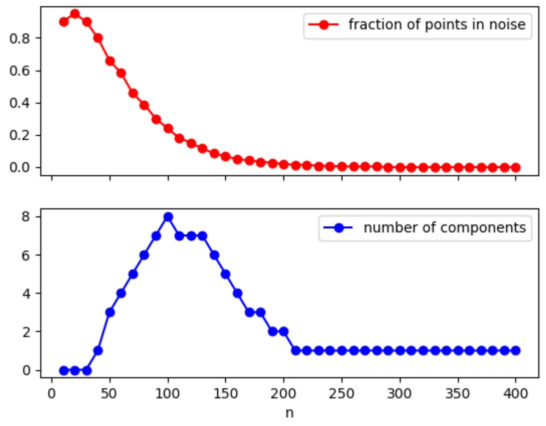
DBSCAN is an algorithm first described by Tarjan[5], but we based our code on the pseudocode of Schubert which is more straightforward, while keeping a complexity of $O(|E| + |V|)$. To achieve this complexity, a stack is used to store the nodes of the cluster currently being visited: when a node is visited, if it is a core node not already visited, we add all the nodes in its ϵ-neighborhood to the stack. The stack is reset once we finished dealing with the stack. Because nodes can enter the stack at most as many times as they have edges leading to them, the complexity is $O(|E| + |V|)$.

3.2 DBSCAN results on 2D plane graphs: finding a heuristic to determine M and ϵ

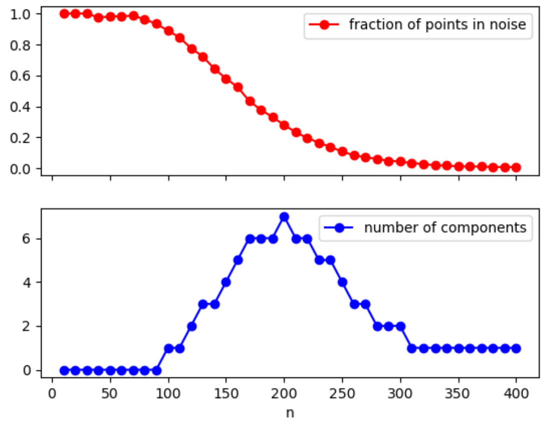
DBSCAN has a significant drawback: it requires the user to choose two hyperparameters M (the minimum number of neighbors a node must have to be considered a core node), and ϵ, the maximum distance at which two nodes can see each other. However, heuristics can help to find these parameters automatically.

To do that, we applied DBSCAN to randomly generating graphs, where nodes are points independently and uniformly taken in $[0, 1]^2$, and where the distance between two points is their euclidean distance. The advantage of this method is that we can generate a lot of graphs with many sizes/densities (they are two sides of the same coins, so we just focused on one number: the number n of points drawn).

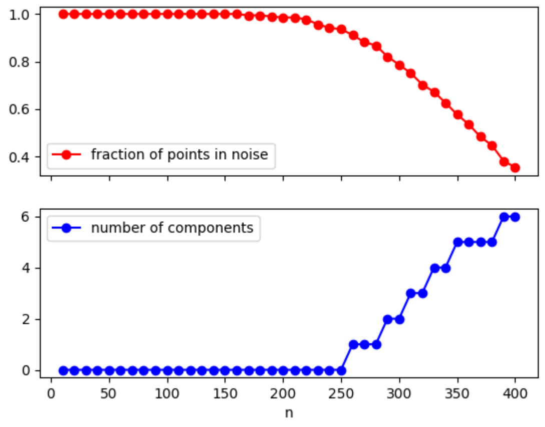
There are multiple ways of measuring what makes a "good" way of partitioning the data into clusters. The INF442 courses detailed this problem for the problem of finding the optimal K in K-Means. However, the problem here is different. First, the hyperparameter that is chosen doesn't explicitly decide what is the number of clusters. But more importantly, we want to find a heuristic that might apply to graphs that don't have a notion of distance built-in. We chose to focus on the following minimalistic desiderata (which is very general): first, a good partition should not have too many nodes classified as noise, and second, a good partition should have at least two clusters. For the 2D graphs at hand, this minimalistic desiderata was enough to find constraints on ϵ and M which were strong enough to find a heuristic.



(a) $\epsilon = 0.05$

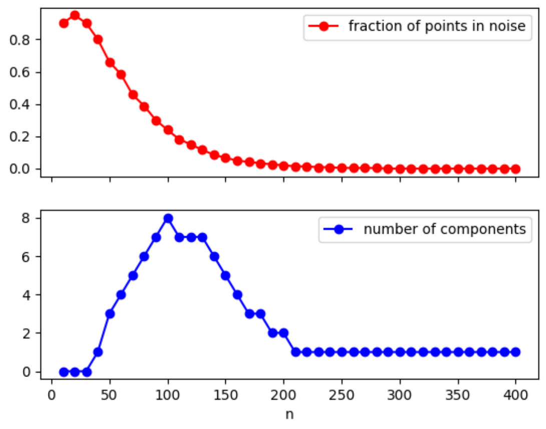


(b) $\epsilon = 0.1$

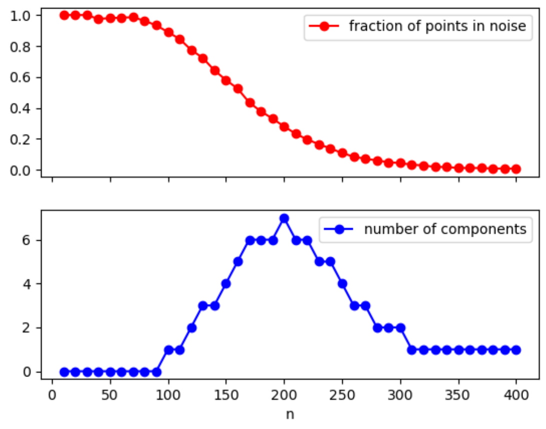


(c) $\epsilon = 0.2$

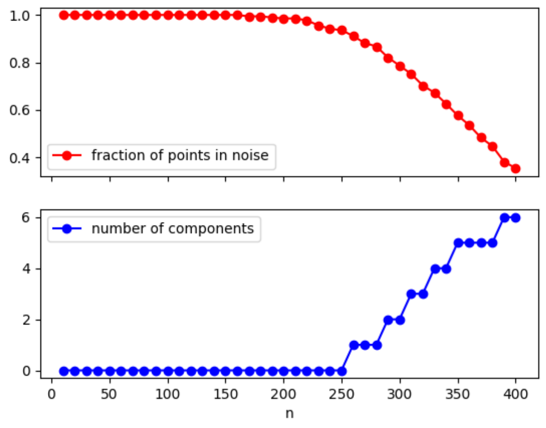
Figure 2: 2D-graphs with $M = 4$, varying ϵ



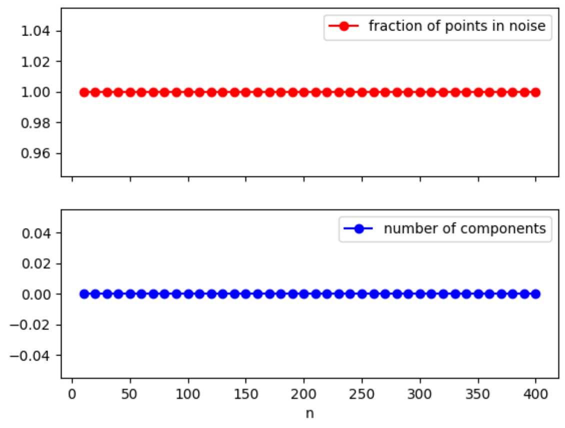
(a) $M = 4$



(b) $M = 8$



(c) $M = 16$



(d) $M = 64$

Figure 3: 2D-graphs with $\epsilon = 0.1$, varying M

The experimental results show that, as one can expect, the ϵ parameter should be of approximately the same order of magnitude as the distance between two neighboring points. Given that we should still find points with at least M points in their neighborhood (but not too many of them, otherwise everything would be connected, except if there were large gaps between clusters), we chose to take ϵ to be equal to the average over every point length between this point and its M-th closest neighbor (when it has at least M neighbors). This choice was confirmed to be good by our experimental results (see Figure 3).

To choose the parameter M, we mainly relied on the experimental results. We noticed that good values of M were of the order of magnitude of $\sqrt{n}$ (this proved to be too large, $\sqrt{n}/2$ works better). This heuristic wouldn't make sense for graphs where each node has only a few neighbors (none of them would be considered core). So our final heuristic more M is $M = \min(\sqrt{n}/2, \overline{\texttt{deg}})$.

The resulting heuristic works well across many values of n in the 2D plane graph setup (See Figure 4).

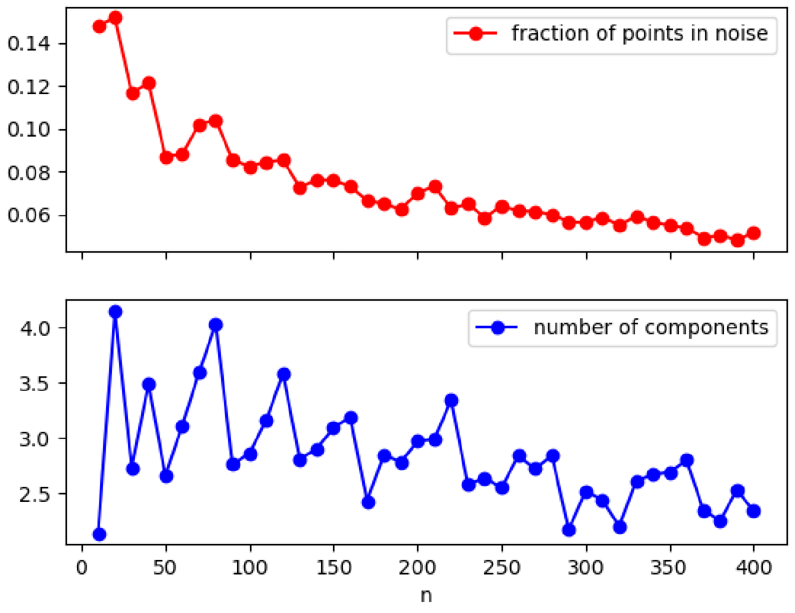


Figure 4: Our heuristic

Further work could find better values of the parameters for our minimalistic desiderata by dichotomy. This should not a problem since the proportion of nodes classified as noise is decreasing with ϵ, and that at a given level of noise, the number of clusters is increasing with M, so that target ranges of "reasonable number of clusters" and "acceptable level of noise" should be reachable. However, this would be much more expensive to run. This technique might face the same problems as K-Means, as the "reasonable number of clusters" should be specified, but it might not be as much of an issue as with K-Means, because the dichotomic search would not impose a number of clusters, but rather push gently the clustering algorithm to a reasonable number (in most real-life scenarios, we know if we want tens of clusters or rather thousands).

3.3 DBSCAN results on a Social circles graph

Finding real-world datasets where the use of DBSCAN was relevant proved a bit tricky, because most of the datasets we found did not have weights (or had weights that were not meaningful to our problem, like star ratings). Thus, we studied DBSCAN results on the Social circles dataset mentioned before[3], which is an undirected connected graph, for which we considered all edges to be of length 1. This made the ϵ parameter irrelevant, because for $\epsilon > 1$, nodes would never have any other node in their ϵ-neighborhood, and for $\epsilon \leq 1$, all connected nodes are in the ϵ-neighborhood.

What we found is that no matter the value of M, there was always one very large component, and no other significant components. When increasing M, the number of nodes in the large component decreases (and the number of nodes classified as noise increases). At a certain point, all nodes are classified as noise.

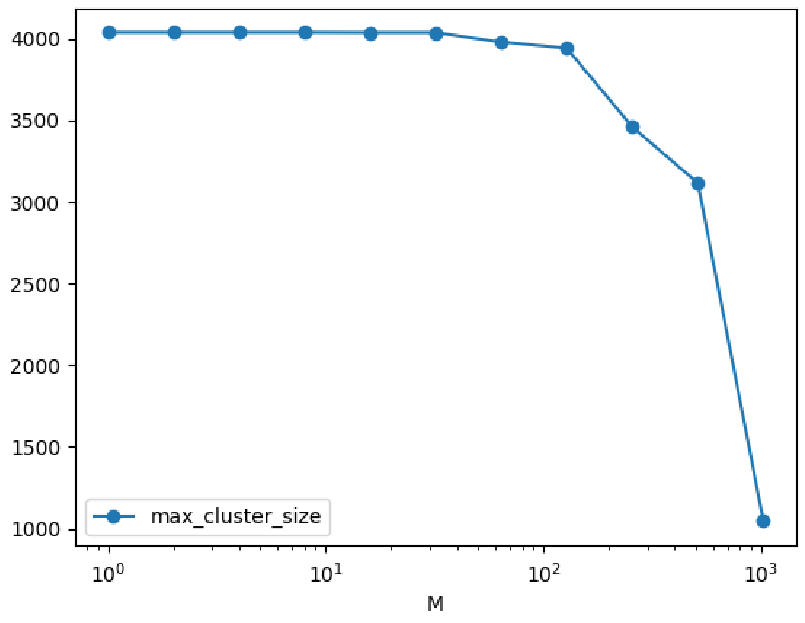


Figure 5: On the social circles dataset, size of the big component vs M

What we can conclude from this analysis is that DBSCAN is not able to find components which could be separated in a meaningful way. It looks like these social circles overlap very strongly: if the value of M is small, small ”circles” of people who don’t know many other people are considered noise, and if M is large, these ”circles” are incorporated into the large cluster because everyone is connected to someone famous in the large cluster.

Further work could make DBSCAN produce more interesting results by setting the length of the link between u and v to $\max(\texttt{deg}(u), \texttt{deg}(v))$: two people who don't have any friends but who are with each other have a bond much stronger than if one of them has thousands of friends.

References

[1] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*, KDD'96, page 226–231. AAAI Press, 1996.

[2] George S. Fishman. Sampling from the binomial distribution on a computer. *Journal of the American Statistical Association*, 74(366):418–423, 1979.

[3] Jure Leskovec and Julian Mcauley. Learning to discover social circles in ego networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[4] Raphaël Rossignol. *Graphes aléatoires.* 2012.

[5] Robert E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1:146–160, 1972.