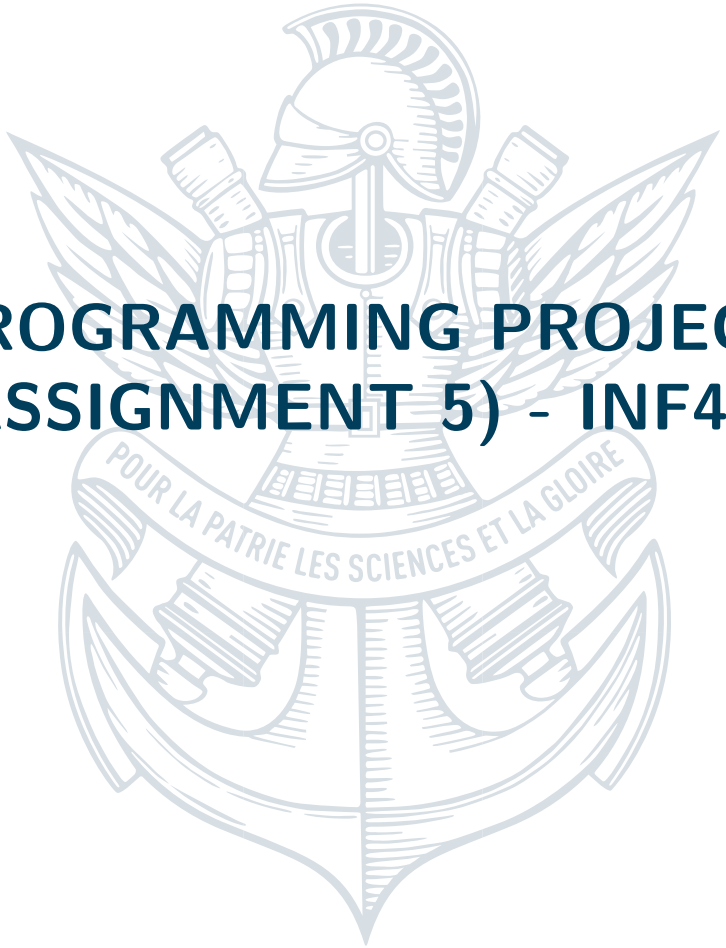


PROGRAMMING PROJECT (ASSIGNMENT 5) - INF442

May 2023

Artur César Araújo Alves
Ksenia Khmelnikova

ÉCOLE POLYTECHNIQUE

IP PARIS

SOMMAIRE

1
OVERVIEW OF THE PROBLEM

Formally, a digraph G with vertices V is strongly connected if and only if for all $u, v \in V$, there exists a (directed) path in G from u to v. In other words, each vertex of G is reachable from any other vertex.

Since strong connectivity for an entire digraph G is a rather strong requirement, we are rather interested in sub-digraphs of G that are strongly connected. We call each induced sub-digraph of G that is strongly connected and cannot be extended to a larger strongly connected sub-digraph a strongly connected component of G. In this project, we have implemented algorithms that utilize strongly connected components and we have run each algorithm on various data sets.

Below we discuss different ways to solve this problem. We also ***highlight*** direct links to our **github repository** containing C++ implementations to the concerned tasks.

2
STRONGLY CONNECTED COMPONENTS

2.1 Kosaraju's algorithm

Firstly, we implemented and tested a sequential algorithm that identifies the strongly connected components of a given digraph, **Kosaraju's algorithm**.

Given a directed graph G, we define its transposition, denoted as G^{-1}, as a graph with an identical set of nodes as G, but with all directed edges reversed compared to G. The algorithm consists of visiting each vertex of a graph G and performing multiple *Depth-first searches* (DFS) to extract a stack S of vertices added in *post-order* traversal, which means that a vertex is added only after all of its out-neighbors have been visited. Subsequently, the vertices in S are processed and an analogous process is applied to the transposed graph G^{-1}, this time assigning the strongly connected components id's to each visited vertex.

As a way of validating the correctness of our implementation, we used it to solve a coding problem related to the subject on the Hacker Earth website :

— **Problem description**
— **Our solution**

Algorithm 1 Kosaraju's algorithm

Let S be empty. Mark all vertices in G as unvisited and unassigned
for each u in G **do**
    if u is unvisited **then** do DFS(u), where DFS(u) is the recursive subroutine :
        Mark u as visited
        for each unvisited child v of u **do** DFS(v)
        Add u to S
Start with $n_{SCC} = 1$
while S is not empty **do**
    Pop u from S
    if u is unassigned **then** do t-DFS(u), where t-DFS(u) is the recursive subroutine :
        Assign n_{SCC} to u
        for each unassigned child v of u **do** t-DFS(v)
        Increment n_{SCC}

2.2 Erdos-Rényi digraph

We consider the Erdos–Rényi (ER) digraph model, which is given a value $p \in [0, 1]$ as well as a vertex set V and constructs a (random) digraph (V, E) by adding each edge $(u, v) \in V^2$ to E with probability p. We wrote a program that generates ER digraphs for a given number of vertices $n \in N > 0$ and a given parameter $p \in [0, 1]$. We take n and p, i.e number of nodes and the probability from the input. Then we create a graph with N nodes without any edges and we add the edges to the graph randomly, take a pair of nodes, and get a random number R. If $R < p$ (probability), we add an edge. We repeat this step for all possible pairs of nodes and then display the whole social network (graph) formed. **Here the code that generates the requested graphs**.

As requested, we played around with different values of n and p. **Our findings** show that for each graph size n, there exists a range of values for p where there are sudden changes in the average number of strongly connected components and the size of the largest strongly connected component.

In figure 1, we display the results obtained by averaging over 10 trials for the two quantities, the average number of strongly connected components, and the size of the largest strongly connected component, across various values of n and p.

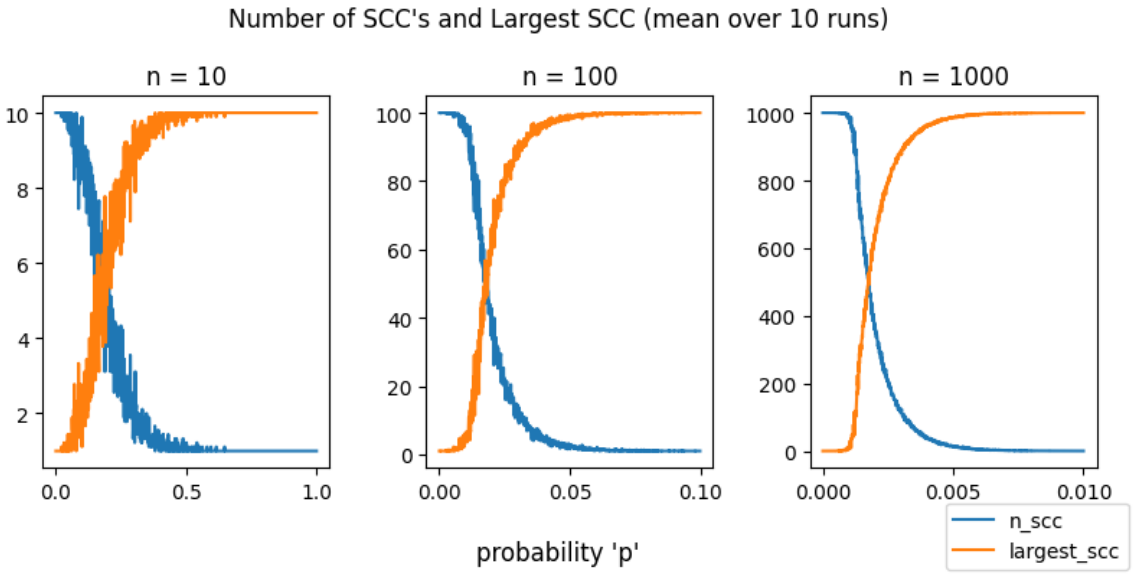


Figure 1 – ER-digraph analysis

3
DENSITY-BASED SPATIAL CLUSTERING

3.1 DBSCAN algorithm

We consider clustering via the DBSCAN algorithm and we implemented and tested the DBSCAN algorithm with random placements of points in a 2-dimensional Euclidean plane as well as with the real-world data sets that are used in the task 1.

First of all, in order to understand the DBSCAN clustering we need to define three types of points that can exist in this algorithm :

1. A point p is **a core point** if at least $minPts$ points are within distance ϵ of it (including p).
2. A point q is **a border point** if there are less than $minPts$ points within distance ϵ of it (including p), but there is at least one core point in its ϵ-neighborhood.
3. All other points are **noise points**.

With these definitions in mind, we can now explain how the DBSCAN algorithm works. The algorithm consists of three main steps :

1. Find the points in the ϵ-neighborhood of every point, and identify the core points with more than $minPts$ neighbors.
2. Find the connected components of core points on the neighbor graphs, ignoring all non-core points.
3. Assign each non-core point to a nearby cluster if the cluster is an ϵ-neighborhood, otherwise assign it to noise.

We implemented **two versions of this algorithm**, one using a breadth-first search (BFS) and the other using disjoint-set union (DSU). Both implementations have a complexity of $O(N^2)$, with the extraction of neighbors of a point being the most computationally expensive step. The BFS implementation has lower overhead and is slightly faster. On the other hand, the DSU implementation allows for parallelization with the creation of threads, which would save a considerable amount of execution time even if we did not implement.

Our implementation has a complexity of $O(|V| \cdot (|V| + |E|))$, which limits the size of the graphs that can be processed efficiently. Specifically, our program can handle graphs $G = (V, E)$ where the sum of the number of vertices and edges is less than 10^4. For such graphs, the program can run within a reasonable time frame of less than 1 second.

3.2 Tests in real world dataset

To conduct **tests using SNAP and CRAWDAD collections**, which mostly contain network-based datasets, we created a distance matrix for every pair of nodes in the graphs. As definition of distance, we chose the length of the shortest path between two nodes in undirected graphs, since it adheres to the axioms of *distance* and therefore we can abstract a metric space.

Here are the results for our DBSCAN implementation **on a data set of autonomous systems AS-733** (numbers of nodes : 6474, number of edges : 13895), and we got the following results for $minPts = 4$:

eps	**Number of classes**	**Numbers of points in the largest class**	**Number of noise points**
1	16	5900	498
2	23	6202	207
3	23	6212	206
4	23	6218	206
5	23	6218	206
6	23	6218	206
7	23	6218	206
8	23	6218	206

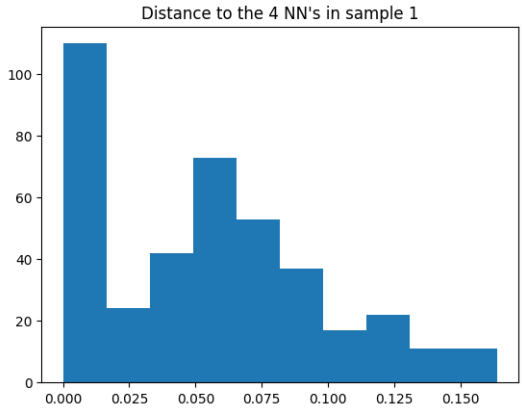
For the dataset in the example above, the average runtime for different values of eps was $0,89s$.

3.3 Tests in random 2-D points

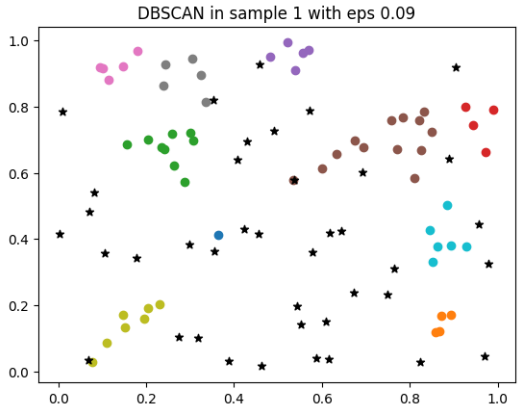
We also **conducted tests** using random 2-D points. We used 100 points generated using a uniform distribution in the unit square. Regarding the DBSCAN parameters, we set $minPts$ to 4 and tried to find a suitable eps.

Our strategy was to extract the distances for each point to it's $minPts$ nearest neighbors and plot a histogram. Afterwards, we try to set eps as the distances where there is the most significant discontinuities in the graphique.

Bellow we show the histogram and the DBSCAN clustering result for one of our generated samples. In this sample, we have set $eps = 0.09$ as there is a significant down gap at this distance. Here are the plots for this sample :



(a) Histogram of distances to nearest neighbors



(b) DBSCAN clustering : noise points are marked as '*'

Figure 2 – Tests in 2-D random generated points