STRONGLY CONNECTED COMPONENTS

Houssam El Cheairi

May 2020

Abstract

We explore here some algorithms to find strongly connected components in graphs and implement a density based clustering method.

1 Task 1

In this part we implemented two algorithms to find the strongly connected components (SCC) of directed graphs. For each algorithm we defined a specific 'Point' class to keep track of the different features the algorithms need to run smoothly.

Tarjan

Tarjan's algorithm is a linear time method to find SCCs in directed graph based on Depth-First-Search routines.

Algorithm 2 Tarjan algorithm

```
for v ∈ V do
    if v.index = None then
        strongconnect(v)
    end
end
```

Algorithm 3 strongconnect method

```
input = v, index, S, cluster
v.index=index
v.lowlink = index
index +=1
S.push(v)
v.onStack = True
for (v, w) ∈ E do
    if w.index = None then
        strongconnect(w,index,S)
        v.lowlink = min(v.lowlink, w.lowlink)
    else
        else if w.onStack then
        v.lowlink = min(v.lowlink, w.index)
    end
end
if v.lowlink == v.index then
    cluster +=1
    w = S.pop()
    while w ≠ v do
        w.onStack = False
        w.label = cluster
    end
end
```

Kosaraju

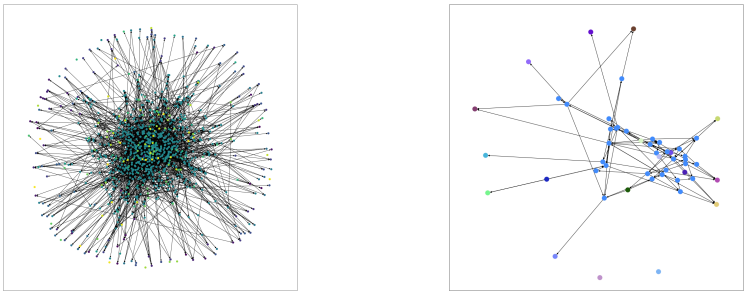
We also implemented Kosaraju; another linear algorithm to find SCCs in directed graphs based on a double DFS. The explicit description of the

algorithm can be found here

Task 2

1.1 Testing with ER graphs

We tested the past two algorithms wih randomly generated graphs *(Erdos-Renyi Graphs)*, by fixing some probability p for each possible edge. Let's denote $D(n, p)$ the ER graph with n vertices and edge probability $p \in [0, 1]$. After experimenting with large values of n we observed that for most values of p the graph either has no non trivial SCC or has one non trivial SCC : the whole graph. Moreover the values of p giving interesting SCC distributions are close to $\frac{1}{n}$. We then run tests with $p = \frac{Cte}{n}$.
We were able to generate nice visuals of the graphs using the *Networks* Python library.



(a) n=1000 , p =0.002 (b) n=50, p=0.04

Figure 1: 2 Figures side by side

Testing with real data

In order to test the SCC algorithm, we used Wikipedia Vote Network and Epinions Social Network datasets available here.

The dataset is a directed graph containing 7115 vertices and 103689 edges. After applying the Tarjan algorithm We found one non trivial SCC of size 1300 whereas the rest of the dataset points are trivial clusters. Apon testing with the second (and larger) dataset, the recursive version of the algorithm fails as the maximum amount of recursive calls is reached quite rapidly, to deal with this issue, we implemented an iterative version of Tarjan's algorithm.

We tested the iterative implementation with the Epinions social network dataset (75879 vertices, 508837 edges), and we found a central non trivial SCC with 32223 vertices, and few other smaller non trivial SCCs of sizes $15, 9, 8, 7, 6, 5, 4, 3, 2$

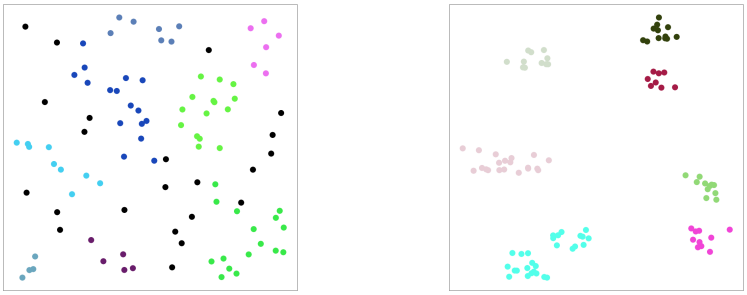
Task 3

DBSCAN algorithm

We implemented the classical version of DBSCAN algorithm as well as a variant based on using edges in order to look for ϵ-neighborhoods.

Testing with random placed 2D points

We tested the DBSCAN algorithm on randomly and uniformly distributed points on the $[0, 1] \times [0, 1]$ unity square. However, since such tests don't fully show the potential of the algorithm we implemented more specific random distributions, by generating random points in small rectangles.



(a) Uniform distribution

(b) Uniform distribution over small rectangles

Figure 2: 2 Figures side by side

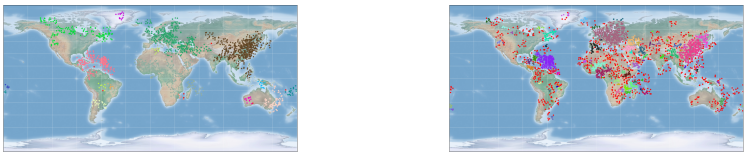
Testing DBSCAN with geographical data

We used open sourced datasets about the Coronavirus confirmed cases and past recorded earthquakes since 2000 with their geographical locations (latitude, longitude). In order to get comprehensive visuals we used the Geopy Python library. Since the provided Covid-19 dataset locations are only precise to a city level, we added to each confirmed COvid-19 case's latitude and longitude a small random noise to obtain distinct dataset points. Note that

the red points are the ones labeled as noise by the algorithm

For better resolution, the original frames are within the zip file.

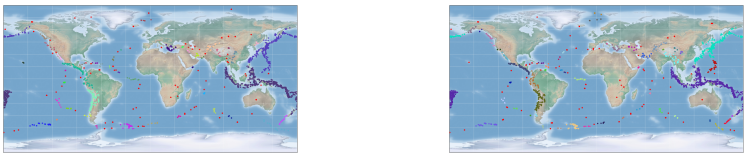
Testing with Covid19 world data



(a) Covid-19 dataset, ϵ=500 (b) Covid-19 dataset, ϵ=200

Figure 3: 2 Figures side by side

Testing with global earthquakes data



(a) Earthquakes dataset, ϵ=500 (b) Earthquakes datasetϵ=200

Figure 4: 2 Figures side by side

Improvements and Heuristics

Data structure

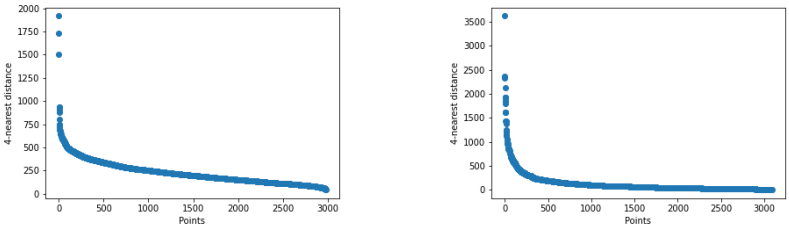
The DBSCAN algorithm requires us to find the points at distance at most ϵ from a candidate core point, however this operation can be very costly in terms of complexity if we use a linear scan of all potential nearest-neighbors. One way to outcome this algorithmic dilemma is to construct a KDTree at

the beginning and use is to send queries about nearest neighbors. The construction of this data structure has complexity $\mathcal{O}(n\log(n))$ and allows us to find the M-nearest neighbors in $\mathcal{O}(\log(n))$ for small values of M which is far better than the initial $\mathcal{O}(n^2)$.

With this efficient implementation we could run DBSCAN algorithm over 1 million data points with an approximate execution time of 45 minutes.

K-NN Heuristic for ϵ choice

The choice of epsilon determines the range or sight limit that we have from each point in the dataset, for a good clustering (good with respect to the intercluster metric) ϵ shouldn't be too big otherwise we'll likely end up with the whole dataset as a unique cluster, and it shouldn't be too small otherwise most points in the dataset will be labeled as noise.
Intuitively, for a fixed parameter M say $M = 4$ for example, we want to have some points with a distance at most ϵ from their 4-nearest neighbor but we also want some other points to have their 4-nearest neighbor at a distance bigger than ϵ.
Assume we have computed the 4-nearest distance for each point in the dataset and denote these distances $d_1, ..., d_N$. ϵ should be chosen such that most noise points have their d_i's sufficiently bigger that ϵ, and many other points (the core points) have their d_i's close from ϵ. Assume we have sorted our distances $d_{\sigma(1)}, .., d_{\sigma(N)}$ in decreasing order, the past observation suggests to choose ϵ as the distance $d_{\sigma(j)}$ with a 'sharpe' positive curve change.



(a) Covid-19 dataset (b) Earthquakes dataset

Figure 5: 2 Figures side by side

The consistency of this idea can be empirically verified with the clusters generated with the test data, we can observe that for ϵ close to the shape curve value on each graph, the clustering was indeed better for each respective dataset.

Probabilistic approach

One of our initial ideas was to add edges between dataset points and only consider them in the ϵ-neighborhood search. The choice of edges need not be uniformly random, more precisely we need a probability distribution that favors points of close distance compared to some fixed reference distance. If we are working with geographical data on the globe we could use:

$$\mathbf{P}(X_1 -> X_2) = \exp(-\frac{||X_1 - X_2||}{1000})$$

where the distances are in Km.

References

[1] Robert Tarjan. *Depth-first search and linear graph algorithms.*. (English) . SIAM Journal on Computing 1(2):146-160,1972.

[2] Martin Ester, Hans-Peter Kriegel, Jorg Sander, and Xiaowei Xu. *A density-based algorithm for discovering clusters in large spatial databases with noise. In:Evangelos Simoudis, Jiawei Han, and Usama Fayyad(eds),*. (English) . Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining (KDD-96). AAAI Press, Meno Park 1996.