Project 442-5
Finding Connected Components

Karaev Nikita, Nikulina Irina

May 21, 2019

1 Strongly connected components

1.1 Introduction

In this project we are going to consider one of the most basic ways to partition a graph, which is finding its connected components and splitting the graph up according to it. We can face the problem of finding connected components while analyzing such networks properties as connectivity, that can give us necessary information about relations between objects. This task can come up, for example, if we want to collect information about communities of people in social network using graph. In this project we will consider directed graphs that are often used in social networks representations and have a various range of real life applications.

For directed graph we will define strong connectivity as follows:
A digraph $G = (V, E)$ is **strongly connected** if it contains a path from u to v for all nodes u, $v \in V$. A sub-digraph of G is a strongly connected component of G if it is strongly connected and maximal with this property. Any digraph has a unique decomposition into strongly connected components, which partitions the set of nodes.

1.2 Algorithm

To find strongly connected components in directed graph we have chosen the **Tarjan's SCC** algorithm, because it is one of the most efficient algorithms for finding the strongly connected components. Its time complexity is $O(|V| + |E|)$ where V is a set of all graph nodes and E is a set of all graph edges.

1.2.1 Tarjan's Algorithm is based on following facts:

1. Depth-First Search search produces a DFS tree or forest
2. Strongly Connected Components form subtrees of the DFS tree.
3. If we can find head of such subtrees, we can print all the nodes in that subtree including head that will be one strongly connected component.
4. There is no back edge from one SCC to another. There can be only cross edges, but they will not be used for the graph processing.

1.2.2 Implementation details

In order to implement the Tarjan's Algorithm we have created a class *Graph* allowing to represent a graph with nodes and edges. It contains such fields as number of nodes n, an adjacency list *adj*, a hashmap *components* which will be filled after the algorithm execution and such methods as $DFS()$, $firstStage()$ and $SCC()$ that are necessary for the implementation of the algorithm itself.

```
class Graph {
        int n, counter, currComponent, biggestCompSize;
        list<int> *adj;
        map<int, list<int>> components;
    public:
        Graph(int n);
        Graph(int n, float p);
        int getBiggestCompSize();
        Graph getTranspose();
        void addEdge(int v1, int v2);
        void DFS(int v, bool visited[]);
        void firstStage(stack<int> &s, int v, bool visited[]);
        map<int, list<int>> SCC();
};
```

1.3 Results

The algorithm finds the right solution for all the proposed social network datasets that we have tested including facebook and twitter graphs. To evaluate the algorithm time performance, we have used randomly generated graphs on 100, 1000 and 10000 nodes. The probability of an edge appearance in each pair of nodes was varied between 0.2 and 0.8 in each stage of this test.

As we know that the execution time for this algorithm depends linearly on the number of graph nodes and edges and as we know that the number of edges can grow quadratically because of the edge appearance probability that is equal to 0.8, we may assume that the execution time will also grow quadratically. The obtained execution time is shown in the table below. This result confirms our assumption.

Number of points	Time, s
10^2	$2,59 \times 10^{-3}$
10^3	$1,61 \times 10^{-1}$
10^4	$3,16 \times 10$

Figure 1: Execution time of Tarjan's SCC algorithm as function of the number of points

2 DBSCAN

2.1 Description

The DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm is used to cluster the data points in some geometrical space with defined non-Boolean distances. One of the most important benefits of this algorithm is that it is able to discover arbitrary shaped clusters and to eliminate statistical outliners (noise). The algorithm is parameterized by two parameters, $\epsilon > 0$, which defines minimal distance between the points that we can call neighbors and $M \in \mathbb{N}$, that corresponds to minimum number of point's neighbors required for it to be a dense point.

2.2 Implementation details

One of the objectives of the work was to implement the DBSCAN algorithm. For its realisation we have created a structure *Node* allowing to represent data points and a class *Points* that consists of data structures and functions that are necessary to implement the algorithm itself. The structure *Node* includes an array of point's coordinates, a label that represents the number of the cluster the point is assigned to and a variable *visited* that equals 1 if the algorithm has already processed the point and 0 otherwise.

```
struct Node {
        double *coord;
        int label;
        int visited;
};
```

The class *Points* stores the information about the points, received as an input in the form of rows of points coordinates, in the array of *Node* structures. The core algorithm is implemented in *dbscan*() method that returns a number of found clusters. This method uses supportive functions *eudclidDist*(), returning the euclidean distance between two points, and *getNumNeighbors*(), returning the list of points situated at the distance at most ϵ from the current point. While executing the algorithm the list of point's neighbors is stored as a list of integers (List<int>) - sequence numbers of points. This structure allows to access, add and delete last and first elements for a constant time. After running the algorithm all points are visited and labeled with the number of cluster they were assigned to or marked as a noise.

```
class Points {
        Node *nodes;
        double eps;
        int minNeigh;
    public:
        int n;
        int dim;
        list<int> getNumNeighbors(int ndNum);
```

```
        bool compare(Node nd1, Node nd2);
        int noiseCount();
        int dbscan();
        double eudclidDist(Node Node1, Node Node2);
};
```

2.3 Tests and results

2.3.1 Performance evaluation

Firstly, 2-dimensional datasets of random generated points were used in order to evaluate the performance of the algorithm. Its time of execution is shown in the table below. This result confirms our theoretical estimation about $O(n^2)$ complexity of our implementation of DBSCAN. Moreover, we can conlude that the largest dataset the algorithm can handle is about 10^5 points.

Number of points	Time, s
10^2	$2,14 \times 10^{-3}$
10^3	$2,09 \times 10^{-1}$
10^4	$1,91 \times 10$
10^5	$2,04 \times 10^3$

Figure 2: Execution time of DBSCAN algorithm as function of the number of points

2.3.2 Choosing parameters

The choice of parameters defines the results of clustering. Setting too small values of ϵ implies searching for neighbours in a circle of small radius, that will lead to a big number of small clusters or even of noisy points. However, if the value of ϵ is too big (bigger than mean distance between clusters), all points could be labeled to belong to the same cluster.

The choice of number of neighbors M also has its considerable impact. Bigger values of M increase the probability that more points will be marked as noise. Let's consider the impact of the choice of parameters on the example of two synthetic datasets that include $N_1 = 500$ and $N_2 = 6000$ points correspondingly. For the first dataset we took values of ϵ equal to 0.03, 0.1 and 1, while the number of neighbors M was fixed to 5. Below we can see the results of clustering using DBSCAN algorithm for different ϵ.

In the first case, the number of neighbors $M = 5$, and $\epsilon = 1$ that is rather big value for this dataset (the order of the distance between clusters), so all points are assigned to the same cluster (Figure 3).

Then we have tried set a significantly smaller value of $\epsilon = 0.03$ while the number of neighbors was always $M = 5$. In this case (Figure 4) every point has less neighbors

in its list (as the area of neighbors search is smaller) and less points can form a cluster, so more points are marked as a noise (marked with violet color) and we observe more small clusters (marked with different colors).

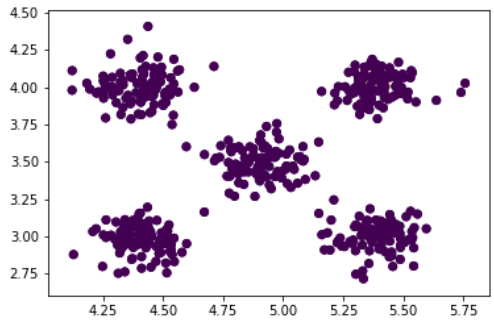

Figure 3: Clustering for $N_1 = 500$, $\epsilon = 1, M = 5$

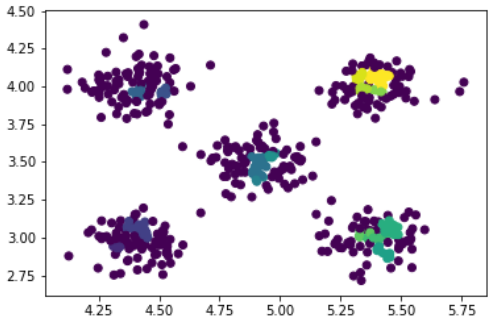

Figure 4: Clustering for $N_2 = 500$, $\epsilon = 0.03, M = 5$

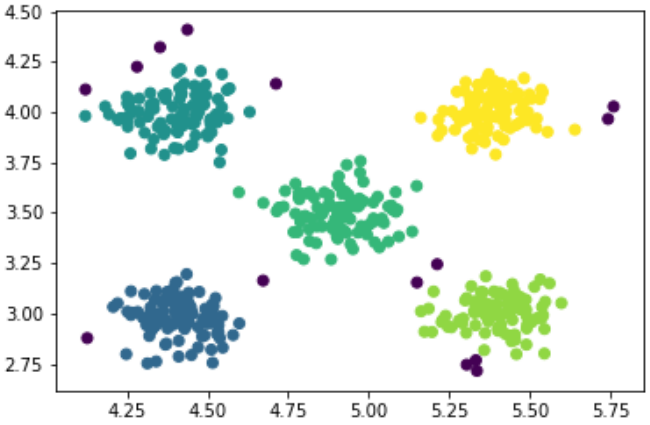

Figure 5: Clustering for $N_2 = 500$, $\epsilon = 0.1, M = 5$

Finally, for the values $\epsilon = 0.1$ and M $= 5$ we can see that almost all points were assigned to five clusters (Figure 5, yellow, green and blues points) with respect to its location and the clustering gives the reasonable result.

As we can notice, the right choice of the parameters, especially ϵ, can significantly improve the resulting clustering. But how the parameters can be chosen besides just trying different values?

One of the ways to do it is to use knn-distances method. Firstly, the minimum number of neighbors M is chosen. Normally it is a number between 3 and 9 depending on the clusters density. Then for each point of dataset the distances to its M nearest neighbors are calculated. Let's consider an example. For the synthetic

dataset with $N_2 = 6000$ points, the distribution of distances to $M = 5$ nearest neighbors of every point is the following:

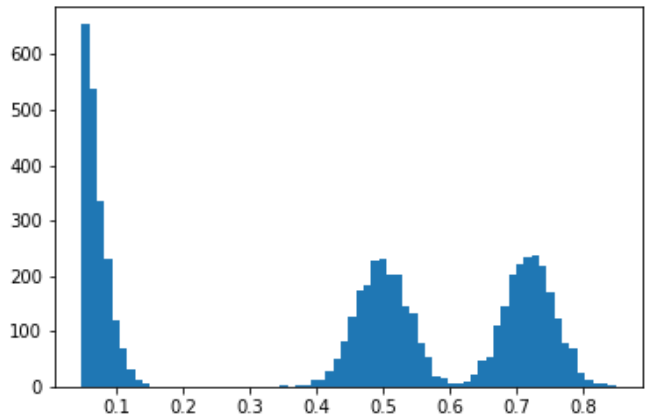

Figure 6: Number of points as a function of mean values of its distances to M nearest neighbors for $N_2 = 6000, M = 5$

As we may notice, the peak of the histogram is located at the mean distance of about 0.025, so the reasonable choice of ϵ will be around this value. The result of clustering using the DBSCAN algorithm with these parameters is shown below (Figure 7).

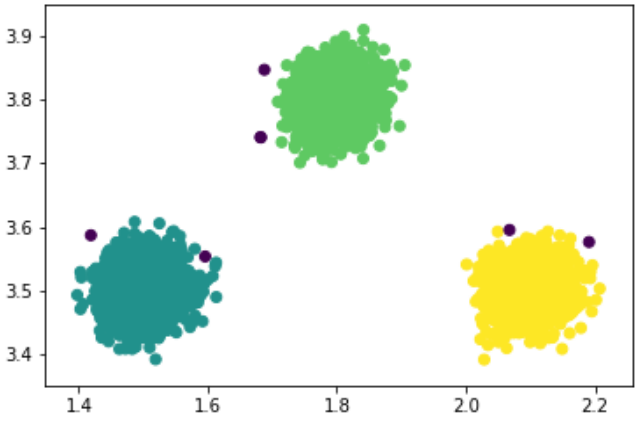

Figure 7: Number of points as a function of mean values of its distances to M nearest neighbors for $N_2 = 6000, \epsilon = 0.025, M = 5$

2.3.3 Real life data set

As a real-life data set we have chosen the famous iris.data dataset,that consists of petal length, petal width, sepal length, sepal width and the name of species for 150

flowers. The first four characteristics allows us to use the euclidean distance between points and the last column is useful to compare the result of clustering with the real grouping. For the ease of visualisation we take two of four columns (sepal length and sepal width) with measurements of flowers. To start the appropriate parameters we chose to use the knn-distances method described above. Below we can see the histogram of mean distance to $M = 4$ nearest neighbors distribution (Figure 8). Its peak gives us an approximate value of $\epsilon = 0.3$, for which we run our DBSCAN algorithm (Figure 9).

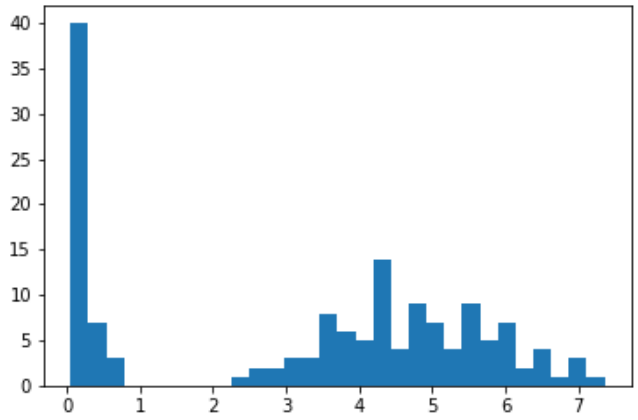

Figure 8: Number of points as a function of mean values of its distances to M nearest neighbors for $N = 150, M = 4$

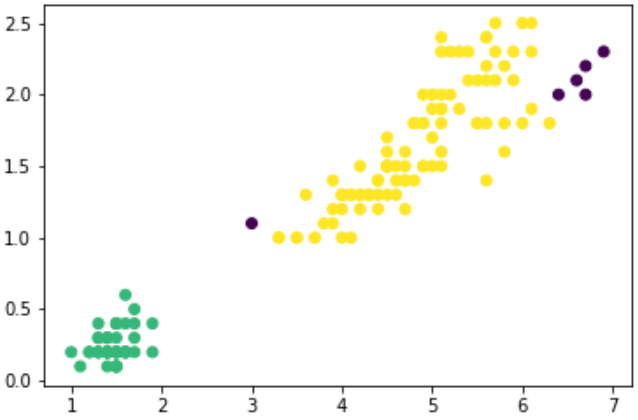

Figure 9: Clusters determined by DBSCAN algorithm for iris.data for $M = 4$, $\epsilon = 0.3$

We can see that the algorithm allows to distinguish only 2 clusters of different forms corresponding to different types of flowers, but looking at the visual representation of points, we can conclude that the result is still reasonable. The number of points classified as a noise is negligible and does not influence the final result.

2.4 Comparison with strongly connected components

To compare the results of the DBSCAN algorithm with the strongly connected components algorithm, it was necessary to find a data that could be transformed to a suitable input for the both algorithms. As it was mentioned earlier, the Tarjan's SCC algorithm takes a graph as an input to find its strongly connected components.

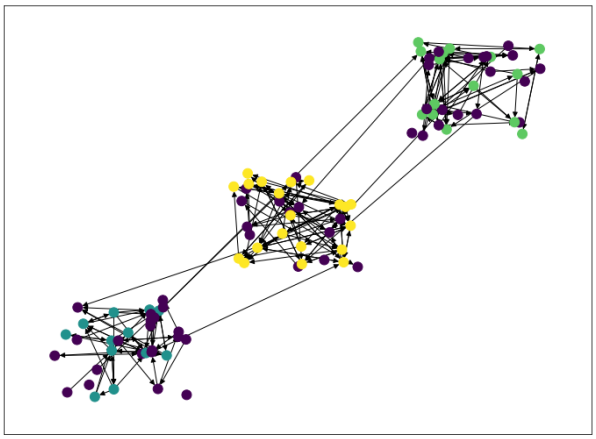

Figure 10: Strongly connected components determined by the SCC algorithm

To test it on the same data with DBSCAN algorithm, we have built a graph based on the input points of DBSCAN as follows. For every pair of points (nodes) we added an edge between them with a probability inversely proportional to the squared distance between the points and the direction of the edge choosing randomly. It allowed us to obtain a directed graph and thus to test both algorithms.

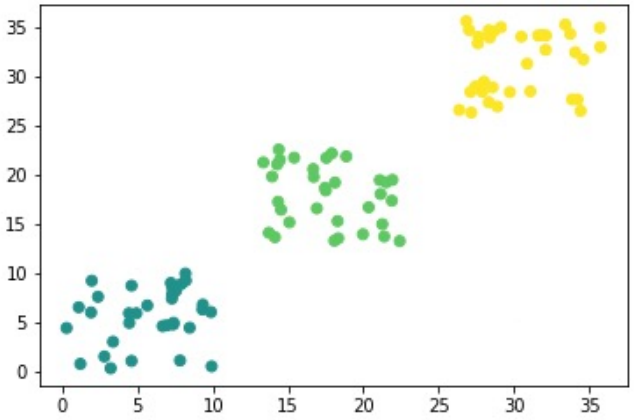

Figure 11: Clusters determined by DBSCAN algorithm

We have generated a dataset of 100 points with three visible clusters and randomly placed points in each cluster. You can see above the results of applying the implemented algorithms to this dataset (Figure 10 and Figure 11). Points belonging to the same connected component and to the same cluster are labeled with the same color. Both algorithms have managed to find connections between the points of the same partition. However, the DBSCAN algorithm have shown better results more accurately determining the clusters, while SCC algorithm has left almost a half of the points out of components (the noise points labeled with violet).

3 Conclusion

In this project we have worked with different approaches to finding and analysing connections between different data types using Tarjan's SCC algorithm and DBSCAN algorithm. Strongly connected components, found by the first algorithm, can give us useful information about graph connectivity. The DBSCAN algorithm allows to successfully determine partitions of points, thus it is a reliable solution for clustering. The both algorithms were tested on random datasets, synthetic and real-life datasets and on the same dataset in order to better understand their differences and common points.