Review of NG-DBSCAN: Scalable Density-Based Clustering for Arbitrary Data

GROUP 10 - algo_miners

SAMAY VARSHNEY - samay@iitg.ac.in - 180101097
PULKIT CHANGOIWALA - changoiw@iitg.ac.in - 180101093
SAI SUMANTH MADICHERLA - madicher@iitg.ac.in - 180101068
KOMATIREDDY SAI VIKYATH REDDY - komatire@iitg.ac.in - 180101036

Contents

SECTION 1: REVIEW OF THE ALGORITHM

ABSTRACT

NG-DBSCAN is an **approximated** and **distributed** density-based clustering algorithm that operates on **arbitrary data** and **any symmetric distance** measure. In this review, we provide an overview of the steps in the NG-DBSCAN algorithm along with its evaluation criteria and applications. The results are obtained through different experiments with **real and synthetic data**, proving the claims about NG-DBSCAN's performance and scalability.

INTRODUCTION

Clustering algorithms are fundamental in data analysis, providing an unsupervised way to aid understanding and interpreting data by grouping similar objects together.
DBSCAN introduced the idea of density-based clustering: grouping data packed in high-density regions of the feature space. DBSCAN has **two important features**: first, it separates "core points" appearing in dense regions of the feature spaces from outliers (noise points) which are classified as not belonging to any cluster; second, it recognizes clusters having arbitrary shapes rather than being limited to ball-shaped ones. But then also, DBSCAN has many limitations which we have discussed in the next section.
Even though several distributed DBSCAN implementations exist: they partition the feature space, running a single-machine DBSCAN implementation on each partition, and then "stitch" the work done on the border of each partition, all these approaches are effective only when dimensionality is low and also are not able to handle any kind of heterogeneous data.

MAJOR BOTTLENECKS SOLVED BY NG-DBSCAN

The proposed algorithm solves the **DBSCAN scalability problem of handling large databases**, the **inconsistency of working with heterogeneous data sets**: through a modified implementation of DBSCAN which is approximated, scalable and distributed, supporting any arbitrary data and any symmetric distance measure. The modified implementation so-called NG-DBSCAN algorithm, in some of the cases even outperforms competing for DBSCAN implementations while the approximation imposes small or negligible impact on the results.

The problem with DBSCAN was that in high dimensional datasets, it partitions the feature space and then merges the spaces which lead to high computational complexity in large datasets. Also when applied to arbitrary distance measures, it requires retrieving each point's ε-neighborhood, for which the distance between all node pairs needs to be computed, resulting in $O(n^2)$ calls to the distance function. NG-DBSCAN on the other hand follows a vertex centric approach in which computation is partitioned by and logically performed at the vertices of a graph, and vertices exchange messages whereby building a neighbour and ε-graph and clusters are built based on the neighbour graph content. This approach enables distribution without needing Euclidean spaces to partition. In most of the existing DBSCAN algorithms where $d >= 6$ or n is high, it is computationally infeasible to run the algorithm either due to memory errors or large time complexity. However the algorithm mentioned in the paper named NG-DBSCAN, is **independent of the dimensionality** of the dataset and is able to run in the time **linear** to the number of data points.

NG-DBSCAN ALGORITHM: AN OVERALL DESCRIPTION AND FLOWCHARTS

The NG-DBSCAN algorithm happens in two phases. Dataset is represented in the form of a graph where each node or vertex is a data point and edges represent the similarity distance measure

between points. In the algorithm we are having ε, $Minpts$, $M_{\max}$, ρ, T_{n}, T_{r}, k as parameters and these are tuned in accordance with the number of data points and as per most efficient computational complexity. NG notation is used for denoting Neighbour Graph here.

(1) **First Phase:** It is implemented through a neighbour graph which converges to k-nearest neighbour graph. It creates the ε-graph and avoids the ε-neighbourhood queries (which were creating high computational cost in DBSCAN).
The ε-neighborhood of a point p is the set of points within distance ε from p.
ε-neighbourhood queries for a given vertex is finding nodes that are within the ε-distance which can take upto $O(n^2)$ if performed naively.
ε-graph nodes are data points where each node's neighbors are a subset of its ε-neighborhood.
At each iteration, all pairs of nodes (x, y) separated by 2 hops in the neighbor graph are considered and if their distance is less than ε, then this edge is added in ε-graph. To speed up the computation, nodes with at least $M_{\max}$ neighbors are removed from the neighbour graph.

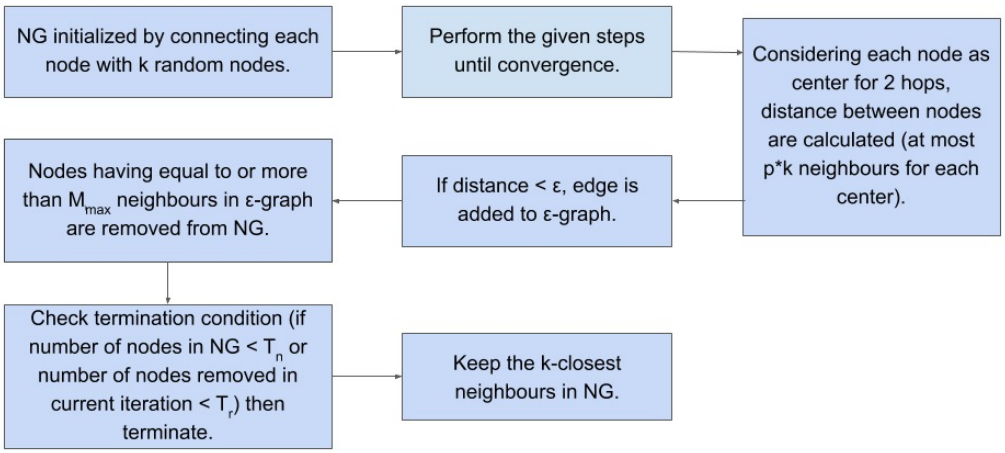

Fig. 1. Overview of Phase 1

(2) **Second Phase:** Second phase takes ε-graph as input to build a clustering and neighbour lookups are performed instead of ε-neighbourhood queries. All nodes are given different roles in the ε-graph.
Core nodes are those having at least $MinPts - 1$ neighbours, Border nodes are those having at least 1 core node as neighbour while Noise nodes are the remaining nodes. Each node coreness is then referred to as $(degree, nodeID)$. This labeling with degree is called coreness dissemination of the graph. Seed of a cluster is called a node with highest coreness. The Propagation forest is created having seed as root for each cluster and from that different clusters are created.

EVALUATION OF THE ALGORITHM

The evaluation of the algorithm was carried out by running it on both synthetically generated and externally used datasets and comparing the results with exact DBSCAN, by comparing its scalability against SPARK-DBSCAN and IRVINGC-DBSCAN. **Quality metrics such as compactness, separation, recall and speed-up** are used to distinguish.
Compactness measures how closely are items in each cluster. Separation measures how well items in different clusters are separated. Recall of a cluster is the fraction of the node pairs that are in

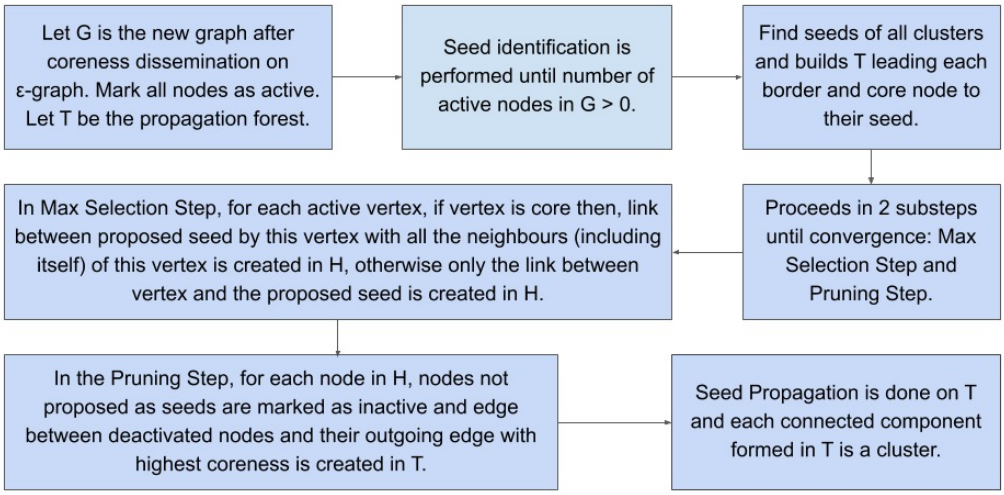

Fig. 2. Overview of Phase 2

the same cluster with that in a reference cluster. Speed-Up metric measures the algorithm runtime improvement when increasing the number of cores dedicated to the computation.
Since computing the above metrics is computationally hard, data points are picked randomly with uniform sampling and averaging independent runs for each data point.
In **2-dimensional space**, clustering quality is compared by checking time taken and recall on different datasets and scalability is measured by taking different dataset sizes and number of cores used.
In **d-dimensional space**, both clustering recall and time taken are compared with the number of dimensions used in different datasets.
In **textual data**, NG-DBSCAN (using both *word2vec* format and *jaro winkler's* edit distance separately) is compared with k-means (for k-means data was converted to word2vec format), using the compactness, separation and time taken metrics.
High compactness, less separation, less time taken was observed in most of the NG-DBSCAN cases in comparison with other DBSCAN algorithms irrespective of the data which proves that clusters are more separated, dense and efficiently computable in case of NG-DBSCAN as seen in the results of the paper.

EXPERIMENT: CODE AND DATASETS

We were not able to find the **Twitter** dataset and the **Spam** dataset used by the author. But we have *mailed* the author asking for the same.

We were only able to find the *implementation* of NG-DBSCAN in Java with Apache Spark framework.

REAL WORLD APPLICATIONS

NG-DBSCAN being a new algorithm (2016), we could not find any real-world applications that claim to use NG-DBSCAN. However, there are ample applications for it's parent algorithm "DBSCAN". And wherever DB-SCAN is used, NG-DBSCAN can be used there. Some applications of DBSCAN (and NG-DBSCAN) are:

- A widespread application of DBSCAN is the geographical clustering, where the goal is to cluster geo points having geo coordinates latitude, longitude. This will be very helpful in the following cases:
  - To *determine* geographical areas that are specific and personal to each user and look at how to build location-based services by extracting users' geographical regions from numerous geolocated events, such as check-ins in restaurants or cafes. Such a system could identify, for instance, areas that a given user typically frequents for dinner outings.
  - When we have too much of *geo-spatial* data, we might need to reduce the size of a data set down to a smaller set of spatially representative points. Here we can use NG-DBSCAN to cluster them down to a smaller dataset.
- Based on previous shows you have watched in the past, Netflix will *recommend* shows for you to watch next. This can be done through NG-DBSCAN clustering of people who have watched similar shows. In fact, this is a very popular use of clustering that we all might be familiar with.
- It can be used to cluster all the emails which are similar to each other in some form which can be used as a symmetric distance measure in the algorithm for clustering.
- It can be used to cluster all the cricketers who are similar in their batting techniques using a d-dimensional dataset where each field in d-dimension is a probability value of playing that kind of shot by the player and taking each player as a data point.

We have not found any ML/Data Analysis Package that includes NG-DBSCAN algorithm.

CONCLUSION

We presented a review of NG-DBSCAN, a novel distributed algorithm for density-based clustering that produces quality clusters with arbitrary distance measures. This allows domain experts to choose the similarity function appropriate for their data, and parallelism can be addressed by designers. We showed an overview of the steps involved in the algorithm and how it is evaluated. We found out that this algorithm has lots of real-life applications and mention some of them in the review report as well.

Our next steps will be to understand the algorithm in-depth and start its implementation. Then we will propose our iterations to make it an incremental clustering algorithm.

UPDATES OF THE DATASET

Alessandro Lulli (Author of the paper) has sent us the datasets.

SECTION 2: RELATED WORK

INCREMENTAL VERSIONS

We didn't find any incremental versions of NG-DBSCAN. But we have found some other incremental versions which are somewhat related to NG-DBCAN and DBSCAN.

(1) **MR-IDBSCAN: Efficient Parallel Incremental DBSCAN Algorithm using MapReduce:** It is also a scalable, density based algorithm to find clusters of arbitrary shapes, size, and as well as filter out noise like NG-DBSCAN does. It also uses the Map Reduce method which NG-DBSCAN follows.

- **Intuition of proposed solution:** In this algorithm, new data points which intersect with old data points are determined. For each intersection point, the new dataset uses an incremental DBSCAN algorithm to determine new cluster membership. Cluster memberships of the remaining points are then updated. R*- tree data structure is used in this algorithm.
- **Results:** Time complexity of this algorithm is less than the original DBSCAN algorithm and it has also dealt with fault tolerance which makes it to compete with NG-DBSCAN.
- **Limitations:** In this algorithm it is difficult to delete clusters incrementally from an existing set of clusters.

(2) **$BISDB_x$: Batch-Incremental Clustering for Dynamic Datasets using SNN-DBSCAN:** $BISDB_x$ is a batch-incremental algorithm based on graph based clustering involving frequently changing dynamic datasets.

- **Intuition of proposed solution:** Incremental version of SNNDB, a graph-based clustering technique is modified to $BISDB_x$ since it was making the process extremely slow when updates are made on larger base dataset. $BISDB_{add}$ is used while adding points while $BISDB_{add}$ is used while deleting points dynamically. It computes K-Nearest Neighbour graph (like in case of NG-DBSCAN), shared nearest neighbours graph, core and non-core points incrementally.
- **Result:** Experimental observations on real world and synthetic datasets showed that $BISDB_x$ are up to 4 orders of magnitude faster than the naive SNNDB algorithm and about 2 orders of magnitude faster than the pointwise incremental method.

(3) **A New Incremental Semi-Supervised Graph Based Clustering:** In the case of semi-supervised learning, before this paper came out, there was no incremental algorithm. This paper introduces a new incremental semi-supervised clustering which is based on a graph of k-nearest neighbor using seeds, namely IncrementalSSGC. In each incremental clustering algorithm, two processes including insertion and deletion for new data points are used for updating the current clusters.

- **Intuition of proposed solution:** Given a k-nearest neighbor graph presenting a data set X, this step uses a loop in which at each step, all edges which have weight less than a threshold θ will be removed. The value of θ is initialized by θ at first step and incremented by 1 after each step. This loop will stop when each connected component has at most one kind of seeds. The main clusters are identified by propagating label in each connected component that contains seeds. The further steps isolate the outliers.
- **Results:** IncrementalSSGC obtains the good results compared with the IncrementalDBSCAN. It can be explained by the fact that the IncrementalDBSCAN can not detect clusters with different densities while IncrementalSSGC does and hence is a competitor for NG-DBSCAN.

VARIANTS

(1) **DENCAST: Distributed Density-Based Clustering for Multi-target Regression:**
DENCAST system, a novel distributed algorithm implemented in Apache Spark, which performs density-based clustering and exploits the identified clusters to solve both single- and multi-target regression tasks (and thus, solves complex tasks such as time series prediction). *Like NG-DBSCAN it is able to handle large-scale and high dimensional data.*

The key features:
- It works on the neighborhood graph. In this way, the algorithm needs only object IDs and their neighborhood relationships (instead of their initial, possibly high-dimensional, representation) and thus it requires limited space resources. We build such a neighborhood graph efficiently from high-dimensional data through the locality-sensitive hashing (LSH) method.
- It is implemented in the Apache Spark framework and it is fully distributed. Therefore, it does not require pre-processing or post-processing steps, usually performed on a single machine. This aspect allows our method to analyze large-scale datasets without incurring in computational bottlenecks.
- The identified density-based clusters can be exploited to predict the value of one or more target variables, by means of a density- and distance-based approach. The result is that the proposed method can be adopted to solve single-target and multi-target regression tasks in a distributed setting.

Algorithm:
- Given the dataset consisting of n labeled objects we first apply a distributed variant of locality-sensitive hashing—LSH to identify an approximate neighborhood graph.
- From this point, the algorithm only uses the neighborhood graph, which can be considered an approximate representation of the objects and their distances, instead of objects represented in the original feature space.
- Our method for density-based clustering then maps each labeled node to a cluster by propagating cluster IDs from core objects through their neighbors. Our approach is iterative and requires a stopping criterion, based on a threshold (*labelChangeRate*), aiming to avoid unnecessary iterations, which would lead to slight changes in cluster assignments. Note: p is a core object if $|N(p)| \geq MinPts$

(2) **MR-DBSCAN: A Scalable MapReduce Based DBSCAN Algorithm:**
MR-DBSCAN consists of three stages: data partitioning, local clustering, and global merging. The first stage divides the whole dataset into smaller partitions according to spatial proximity. In the second stage, each partition is clustered independently. Then the partial clustering results are aggregated in the last stage to generate the global clusters.
Let S_u denote the minimum bounding rectangle (MBR) of all the input points in ***DB***. During data partitioning, we divide S_u into non-overlap sub-rectangles. All the input points in ***DB*** that fall into or close to a rectangle form a partition. The local clustering stage performs sequential DBSCAN for each data partition separately and save the local clusters as intermediate results. Sequential merging of local clusters becomes inefficient when dealing with very large datasets. To address this problem, a parallel algorithm is proposed to ensure the scalability of this stage.

(3) **KNN-DBSCAN: DBSCAN in High Dimensions:**
To enable density clustering of high-dimensional datasets we propose a DBSCAN algorithm

that uses an approximate k-Nearest-Neighbor graph (k-NNG). For the exact k-NNG, an edge E_{ij} between vertices i,j exists if and only if p_j is in the k-nearest neighbor list of i.
Notice that k-NNG is a directed graph—in contrast to ϵ-NNG. The main reason k-NNG is used is that we can probably control the work and memory complexity. That is, memory requirements for k-NNG always remain $O(n * k)$, whereas for ϵ-NNG can explode to $O(n^2)$. Instead of running out of memory or creating severe load imbalance as is often the case with ϵ-NNG, k-NNG only suffers from a loss in accuracy. But convenient as k-NNG is, DBSCAN requires a symmetric ϵ-NNG. To circumvent this, kNN-DBSCAN is introduced that uses the k-NNG.

SECTION 3: CODE ARCHITECTURE

CLASS DEFINITIONS AND DATA STRUCTURES

In terms of data structures, **graphs**, **lists** and **priority queues** are used. Some of the classes used are listed below.

```
class Node {
    id              -   represents node ID
    degree          -   represents number of edges connected to this node
    coreness        -   represents (id, type) of node
    coordinates     -   represents the data point from our dataset
    type            -   represent whether Node is core, border or noise
    Node(id){       -   constructor to initialize Node
        this.id = id
        this.type = none
    }
}
class Graph {
    int N                           -   Total number of nodes
    set<Node> active                -   List of active nodes
    vector<Node,int> edges[N]       -   Adjacency List
    Graph(int total_nodes) {
      this.active = {u∀u ∈ N}       // make all nodes as active
      this.N = total_nodes
    }
    void remove_node(Node ID)     // removes node from graph
    void add_node(Node ID)        // adds node in graph
    void remove_edge(x, y, w)     // removes edge between node x and y with weight w
    void remove_edge(x, y)        // removes edge between node x and y
    void add_edge(x, y)         // adds edge between node x and y
    void add_edge(x, y, w)        // adds edge between node x and y with weight w
    vector<Node> neighbours(x)    // returns all neighbours of node x
}
class Parameters {
   k            -   represents degree of each node in neighbour graph
   Minpts       -   each core node is having degree at least Minpts − 1
    T_n         -   limits number of nodes in NG for termination
    T_r         -   limits number of removed nodes in current iteration in NG
    M_max       -   used to reduce NG in phase-1 to reduce computation
    ρ           -   limits nodes for which 2 hop distance is calculated in NG
    iter        -   used to achieve convergence condition
   Parameters() {
      // initialise all with default values unless explicit values are given
    }
}
```

PROCEDURES

Here are the procedures which will be used in the implementation of NG-DBSCAN algorithm. In Phase 1, the main function to be called is *ϵ-graph_construction* which will use other procedures listed below.

Phase 1:

```
// creating ϵ-graph
Graph ϵ-graph_construction(int total_points){
   Graph ϵG(total_points), NG(total_points)
   Random_Initialisation(NG)    // initialising each node with k random edges
   delta = 0                    // number of nodes removed in current iteration
   while(i < iter and not Terminate(NG, delta)){
       Reverse_Map(u, NG) ∀u ∈ NG
       Check_Neighborhood(u, NG, ϵG) ∀u ∈ NG
       Reduce_NG(u, NG, ϵG) ∀u ∈ NG
        i++
    }
    return ϵG
}
void Reverse_Map(Node n, Graph G) {
    // Making the graph undirected
   for each u in G.neighbours(G) {
      w = distance(v, u)
      G.add_edge(u, v, w)
   }
}
// checking neighbour graph and updating ϵ graph
void Check_Neighborhood(Node n, Graph NG, Graph ϵG) {
    N = Random selection of at most ρk nodes from NG.neighbours(n)
   for each vertex v in N {
      for each vertex u in N\{v} {
         w = distance(u, v)
         NG.add_edge(u, v, w)
         if(w <= ϵ)
              ϵG.add_edge(u, v, w)
        }
    }
}
// checking termination condition
bool Terminate(Graph NG, int delta){
   if(|NG.nodes| < T_n and delta < T_r){
      return 1
   }
   else {
      return 0
   }
}
// reducing neighbour graph
void Reduce_NG(Node u, Graph NG, Graph ϵG){
```

```
   if(|εG.neighbours(u)| >= M_max){
       NG.remove_node(u)
       delta = delta + 1
   }
   vector<Node> l = NG.neighbours(u)
   remove from l the k edges with smallest weights using priority queue
    for(u, v, w) in l {
        NG.delete_edge(u, v, w)
   }
}
```

Phase 2:

In Phase 2, main function to be called is *Discovering_Dense_Regions* which will further call other procedures listed below.

```
// finding all the clusters
Graph Discovering_Dense_Regions(Graph ε-NN, int total_nodes){
   // Make all nodes active
   Graph T(total_nodes)
   G = Coreness_Dissemination(ε-NN, total_nodes)
   while G has active nodes {
      Graph H(total_nodes)
      // Seed identification process
      Max_Selection_Step(n, G, H) ∀n ∈ G
      Pruning_Step(n, T, H, G) ∀n ∈ H
   }
   return Seed_Propagation(T)
}
// return maximum core node from the list
int Max_Core_Node(vector<Node> list){
   // returns node from list having maximum coreness value
}
void Max_Selection_Step(Node u, Graph G, Graph H){
   vector<Node> NNg = G.neighbours(u)
   int u_max = Max_Core_Node(NNg ∪ {u})
   if(u.type != core){
       H.add_edge(u, u_max)
    }
    else {
      for each node v in NNg
           H.add_edge(v, u_max)
    }
    H.add_edge(u_max, u_max)
}
// in pruning, nodes not proposed as seeds(those with no incoming edges) are deactivated
void Pruning_Step(Node u, Propagation_Tree T, Graph H, Graph G){
   vector<Node> NNh = H.neighbours(u)
   int u_max = Max_Core_Node(NNh)
   if(u.type != core){
       G.remove_node(u)
```

```
       T.add_edge(u_max,u)
    }
    else {
      for each node v in (NNh \ {u_max}){
          G.add_edge(v, u_max)
          G.add_edge(u_max, v)
        }
        if u not present in NNh {
          G.remove_node(u)
          T.add_edge(u_max,u)
        }
    }
}
// convert graph to identify all nodes and return it
Graph Coreness_Dissemination(Graph ϵ-NN, int total_nodes) {
   // Here we identify the core, boundary and noise points of the input ε-graph.
   Graph G(total_nodes)
   G = ϵ-NN
    for each node v in G.nodes {
      if(|G.neighbours(v)| >= Minpts − 1)
         v.type = core // Make it a core node
   }
    for each node v in G.nodes \ {core nodes} {
      if there is at least one core node in G.neighbours(v)
         v.type = border // Make it a border node
   }
    for each node v in G.nodes \ {core nodes} \ {border nodes} {
      v.type = noise    // Mark them as noise points
      G.remove_node(v)  // We need to remove all the noise points from G
   }
    return G;
}
Graph Seed_Propagation(Graph T) {
    a = List of all seeds
    for all nodes v in a {
      Perform Depth First Search on graph T with v as input.
      // Here we will get all the nodes belonging to the cluster of v
    }
    // The final output will be a list of lists where each list corresponds to a separate
        cluster
}
```

In the Seed_Propagation function, we use the already known **DFS (Depth First Search)** algorithm to find all the reachable nodes from a given node. The list of all seeds will be obtained at the end of Seed_Identification step. At the end of the last Pruning Step, the H graph contains only seeds and all the seeds would be present in the H graph. Therefore DFS can be performed with all the seeds. The number of seeds will be the number of clusters formed.

We will require 3 Graphs for Phase 2: Graph G, H and T. Here G and H don't have any precise name as their structure keeps changing with time. The aim of G and H is to ultimately obtain the "T" graph. The "T" graph is the Propagation Forest.

The “T” graph contains several trees where the nodes of each tree are considered to form one cluster and the root of the tree will be the seed of the cluster(core node with maximum coreness which can be thought of as representing the whole cluster).
After obtaining the “T” Graph, we call the Seed_Propagation function which will recover the clusters from the “T” Graph.

SECTION 4: CODING OF STATIC ALGORITHM, PART 1

CODING PHASE

Since we didn't find any existing implementation of the algorithm in C++ or any preferable language, the **major work which we have done is writing the complete algorithm code in C++.**
We also tried much to reach to the authors regarding the existing implementation of C++ code and other resources but we didn't receive any c++ implementation from their side. They have given Scala implementation of the algorithm, but we thought of coding on our own, as understanding Scala would have taken more time.
Mails and Scala Code

CODE FILES

Currently our code clusters the data set consisting of 2-dimensional points. We have generated some random data points using python in a file named points.txt and these points are taken as input points for the C++ algorithm code. Then using our C++ algorithm, we have created a lists of list of points inside C++ program where each list will be a seperate cluster. Then we have written these points in a cluster.txt file and then this cluster.txt file will be used by a clusters_generator.py file which will create clusters in 2-dimension. We have considered for now the distance between 2 points as the euclidean distance between them.

Code
Here is the explanation of the conventions of the files used while writing the code:

- **classes.h -** contains all the used classes in the algorithm.
- **phase1.cpp -** contains the phase-1 code which will be used to create ϵ-graph.
- **phase2.cpp -** used to create the propagation tree and list of clusters.
- **epsilon_graph.txt -** represents the epsilon graph used in the algorithm.
- **propagation_tree.txt -** represents the propagation tree generated by the algorithm.
- **points.txt -** contains the randomly generated points used as input in phase2.cpp.
- **clusters.txt -** contains the lists of list of clusters.
- **galaxy_type_dataset.py -** contains the python code to generate the random points.
- **clusters_generator.py -** plots the clusters in 2-dimension in different colours using clusters.txt.

First, galaxy_type_dataset.py is executed which will take as input **number of points**. Then phase2.cpp will execute. Then clusters_generator.py will generate clusters (different colours denoting different clusters generated by our algorithm) using matplotlib library.
Here are some instructions to run the codes:

- **galaxy_type_dataset.py -** *python3 galaxy_type_dataset.py*
- **phase2.cpp -**
  - Compilation: *g++ phase2.cpp*
  - Execution: *./a.out*
- **clusters_generator.py -** *python3 clusters_generator.py*

EVALUATION

Currently we have evaluated it for 2-dimensional points with galaxy type dataset only. Currently our algorithm is able to run for the number of points upto 100000. It can also run above these number of points but since the C++ compiler is not able to support much memory, hence we are

getting segmentation fault (memory limit exceeded) for $number of points > 100000$. Also for higher K, it is not supporting. Plot Graphs For Our Input Dataset: Plot

In *clusters_generator.py*, we have the code generated for one of the following clusters.

CLUSTER PLOTS

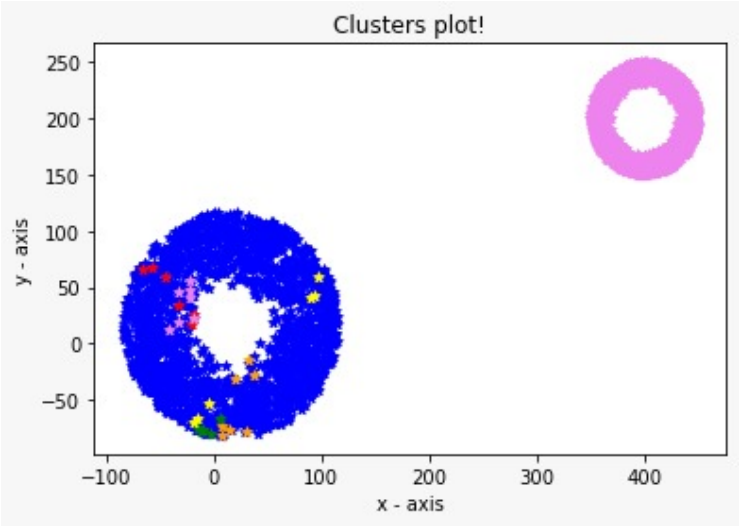

Fig. 3. Clusters With K = 10

We tried changing the parameters used in the code to see the effect of change on the cluster quality.

For the following points distributions, we found the cluster for k = 3 and k = 10 respectively. After analysing the plots, we found that with k = 10 we are getting better results. *Left Image is with Clusters k = 3 and Right Image is with Clusters k = 10.* Here T_n, T_r are very small.

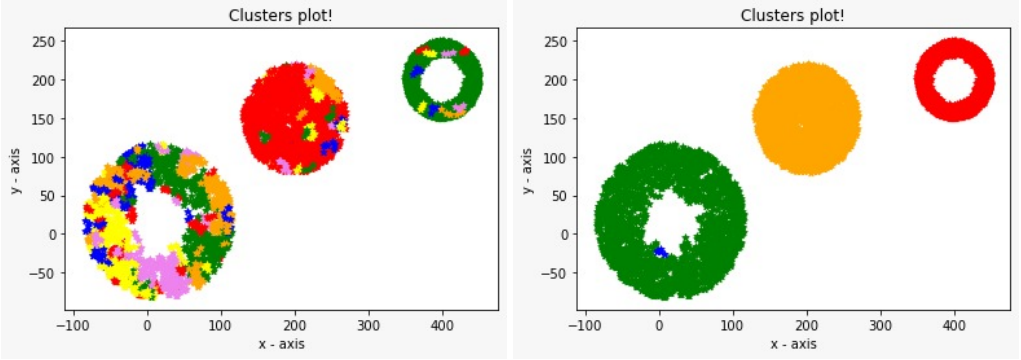

Fig. 4. Clusters With K = 3

Fig. 5. Clusters With K = 10

For the following points distributions, we changed T_n, T_r majorly.

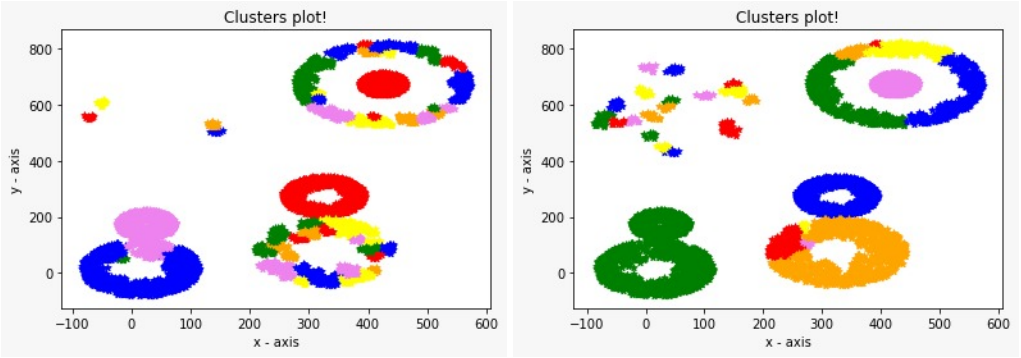

Fig. 6. Clusters With M_{max} = 20, $epsilon$ = 10, T_n = 0.01*n, T_r = 0.001*n

Fig. 7. Clusters With M_{max} = 10, $epsilon$ = 12, T_n = 0.001*n, T_r = 0.0001*n

FURTHER WORK

For the next week we are planning to generalise the distance computations. We will use different evaluation metrics which we have mentioned earlier in our report to test our algorithm and compare them with mainly DBSCAN (and different variants of it if possible). We also haven't tuned our parameters properly till now so we will also work upon this. We have not completed the testing of the algorithm on the datasets used by the author in the publication and will work upon that also. We will also try to create it as arbitrary symmetric as much as possible.

Apart from these we will think which part of the code needs to be modified to convert it into an incremental algorithm.

SECTION 5: CODING OF STATIC ALGORITHM, PART 2

Here is the link to the github repository of the NG-DBSCAN algorithm: Link.

CODE WALK THROUGH

Our algorithm generates density based clustering for given data set using NG-DBSCAN algorithm.

In phase2.cpp, the algorithm is running from the main function. First we asked whether user wants to change the default parameters or not.

```
User@Pulkiitttttttt MINGW64 /e/.Study/Sem 6/Datamining/NG-DBSCAN/NG-DBSCAN Code (
main)
$ python dataset_generator.py
Enter number of points/text sentences:
1000
which dataset you want to generate
 Enter 1 for moon dataset
 Enter 2 for blob dataset
 Enter 3 for circle dataset
 Enter 4 for text dataset
1

User@Pulkiitttttttt MINGW64 /e/.Study/Sem 6/Datamining/NG-DBSCAN/NG-DBSCAN Code (main)
$ g++ phase2.cpp

User@Pulkiitttttttt MINGW64 /e/.Study/Sem 6/Datamining/NG-DBSCAN/NG-DBSCAN Code (main)
$ ./a
Want to change the default parameters?
Enter 1 for Yes and 0 for No
0


User@Pulkiitttttttt MINGW64 /e/.Study/Sem 6/Datamining/NG-DBSCAN/NG-DBSCAN Code (main)
$ python clusters_generator.py
```

About points.txt: We are assuming that all the input data (textual, dimensional or randomly generated) will be present in points.txt. First line will contain dataset type (text or non-text), total number of points and dimension of the data (we assumed 0 dimension for textual data for simplicity). In rest of the lines, each line will represent a point.
Then we read all the points from points.txt depending upon whether our dataset is textual or not and call epsilon_graph_construction function which will return ϵ-graph using phase1.cpp and we will use this graph as parameter to Dense_Discovery_Regions function and it will return propagation forest (T).
In Seed_Propagation function, clusters.txt and numbered_clusters.txt are generated.
About clusters.txt: All the clusters generated will be present in clusters.txt. First line will contain number of clusters generated and dimension of dataset. Then in further lines, for each cluster, number of clusters present in current cluster is in seperate line followed by each point/sentence of that cluster in seperate lines.
About numbered_clusters.txt: All the clusters with the number/index of each point will be present in numbered_clusters.txt.
About classes.h: We defined different important variables for our convinience here. dataset_type variable will denote the type of dataset taken (textual or non-textual). In case of non-textual dataset, coordinates vector will store all the points. coordinates[i] will denote ith point of our dataset. In case of textual dataset, sentences will store all the sentences. sentences[i] will denote ith point of our dataset. node_from_id map will store each index of point to the actual point of Node datatype which contains all information about that point.

COMPUTATIONS OF THE ALGORITHM

We used matplotlib in generate_clusters.py to generate the clusters for 2 and 3 dimensional datasets.

Apart from this, we used different metrics of seperation, compactness and recall to calculate how accurate our algorithm is working. metrics_main.cpp will calculate the seperation, compactness and recall using clusters.txt for the dataset of points present in points.txt.

For the textual data, we are using jaro winkler's algorithm (as used by the author of the paper) to calculate distance between two points. jaro_winkler_distance.cpp contains the code to calculate distance between 2 textual points and it is used in phase2.cpp.
We have also tried to compare our algorithm with DBSCAN algorithm using the above metrics. We ran the same dataset on both the NG-DBSCAN and DBSCAN and compared their metrics.

TOY DATASET

We have taken 15 two-dimensional points as an input of the toy dataset. These points are present in file named toy_dataset.txt.
The points of toy dataset are listed below. First line represents that it is non textual data, has 15 points and is of 2 dimension. We took 2-dimensional for our simplicity in explaining.
The 1st column represents x-coordinate while 2nd column represents y-coordinate of the points. ith row represents ith point.
In phase2.cpp, we entered the following values of parameters for toy dataset.

```
non_text 15 2
-11.004161206558486 -10.24530372973464
3.1768184552599394 -9.006254703405759
-8.357174823998038 -8.854391074446012
-8.435048195710767 -7.806762293952905
-7.393981404723236 1.028497940990529
-6.465388143821774 0.4870595871129323
1.7378505231503554 -9.575283879631659
1.1437903454896514 -9.697724553571327
-6.9145102974713275 3.033882068724556
-6.570426782705661 1.1091458038063127
2.6502349385029955 -8.78415509686841
-9.008866366830741 -8.939591350340011
-8.198548337708637 -9.689507438131614
2.294976018616592 -10.504253545393299
-8.244677296600988 3.2921021434741
```

Fig. 8. Toy Dataset Points

The red edges between (i,j) indicates the edge between the ith and jth point in the epsilon graph.

```
Want to change the default parameters?
Enter 1 for Yes and 0 for No
1
Enter Parameters(If you want to keep default value then enter -1
Enter x for Tn(Tn = x*n)
-1
Enter x for Tr(Tr = x*n)
-1
Enter k
2
Enter Mmax
3
Enter p
2
Enter iter
10
Enter epsilon
1.5
Enter Minpts
3
```

Fig. 9. Toy Dataset Parameters

Each point in the epsilon graph is connected to atleast $M_{\max}$ more points in their neighbour (as seen in the figure) which are within ϵ distance here.
The propagation tree is having 2nd, 4th, 6th points as seeds of different clusters and hence 3

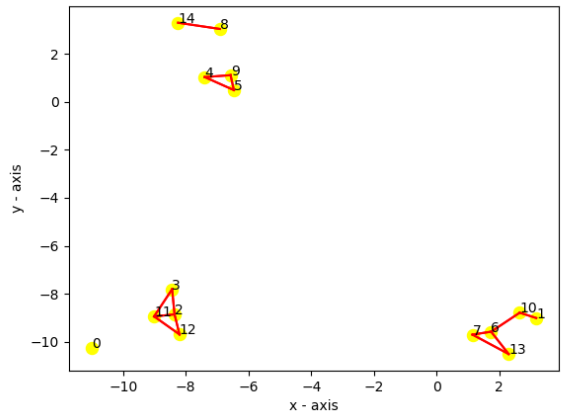

Fig. 10. Toy Dataset Epsilon Graph

clusters are formed.
We found that the points 0th, 8th and 14th are noise since no core node is present in their neighbour within ϵ distance, while 1st point is border since only one core node (10th) is present in it's neighbour, while others are core points. In the toy dataset identification of clusters, black points

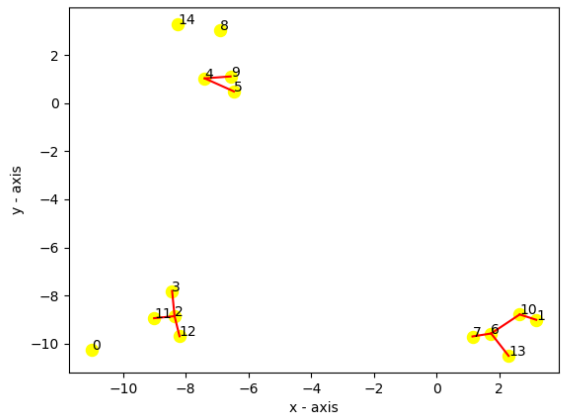

Fig. 11. Toy Dataset Propagation Tree

refer to the noise points while green points corresponds to 1st cluster and blue corresponds to 2nd cluster and red corresponds to 3rd cluster.

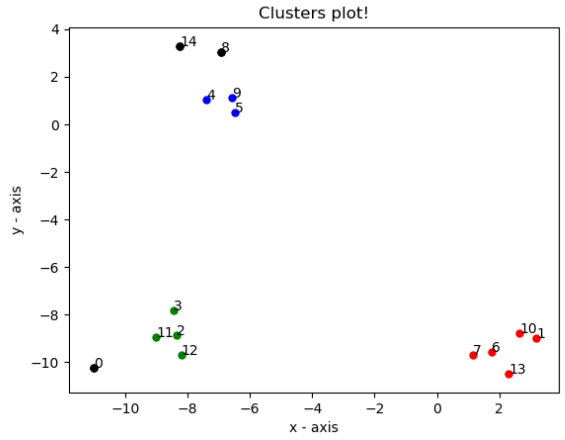

Fig. 12. Toy Dataset Clusters Identification

TEXT DATA COMPATIBILITY

We have added text data compatibility in the algorithm. For this we are using Jaro-Winkler's Distance Algorithm.

Using Jaro-Winkler's Distance between two strings, we define their distance to generate their epsilon graph. Then with the epsilon graph we generate the clusters for the data set in similar fashion as with Euclidean distance measure.

In dataset_generator.py we ask user, the type of data to generate. In the points.txt(our dataset file) we are adding "text" or "non_text" to differentiate between two types of data set. We used the fact in comparison of strings that, the lower the Jaro–Winkler distance for two strings is, the more similar the strings are.

SECTION 5: PROPOSED INCREMENTAL ALGORITHM, ITERATION 1

INTRODUCTION

Many real-world applications such as search engines and recommender systems are expected to work over dynamic datasets. Such datasets undergo frequent changes where some points are added and some points are deleted. A naive method to get exact clustering over the changed dataset is to run the clustering algorithm again. Most of the computation in reclustering is going to be redudant. This problem becomes more severe with increase in frequency of updates to dynamic datasets.

To overcome this problem, we are proposing an incremental version of NG-DBSCAN algorithm, for which we have proposed static version earlier, the incremental version will process frequent updates to dynamic datasets efficiently by avoiding redundant computation. Our incremental NG-DBSCAN will yield significant speed-up factors over NG-DBSCAN even for large numbers of daily updates.

DESCRIPTION AND WORKING OF THE INCREMENTAL VERSION

Our incremental version will happen in 2-phases like in case of static NG-DBSCAN. Sets of updates are processed one at a time without considering the relationships between the single updates.

(1) **Identification of nodes during addition and deletion:** In the static version of NG-DBSCAN, we identify each node as core and non-core (border and noise points). Here also we will maintain each node coreness with it's degree. To do this, we will maintain a set of points I, which will contain all the points for which we need to gather some information to classify them as core or non-core. This set will change during insertion and deletion of points. Here also we are using the convention that the points having atleast *Minpts* neighbours are core nodes.

- **Addition of a point:** Denote the point to be added as 'p'. While adding 'p' to our existing dataset, some of the existing points from dataset can change from non-core to core. So we insert 'p' to the set I. Those points which are having their degree as *Minpts-1* are also added to the set since there is a chance that they become core due to the addition of 'p'. Our incremental algorithm analyses only the region affected by the insertion of point. The region affected by addition will be ϵ-neighbourhood of point p and all points density reachable from any point in $N_\epsilon(p)$. Here $N_\epsilon(p)$ denotes the ϵ-neighbourhood of p.

  Mathematically: $Affected_D(p) = N_\epsilon(p) \cup \{ q \mid \exists o \in N_\epsilon(p) \wedge q >_{D \cap p} o \}$
  Here, $x >_D y$, denotes y is density reachable from x w.r.t data set D.

  We generate a set $UpdSeed_{Ins}$ which contains all core nodes in the epsilon neighbourhood of newly identified core nodes. (As adding point *p* can make some points core nodes).

  Mathematically: $UpdSeed_{Ins}$ = {q | q is a core object in D ∪ {p}, ∃ q': q' is core object in D ∪ {p} but not in D and $q \in N_\epsilon(q')$}

  Efficient computation of $N_\epsilon(p)$:
  – For all the points 'u' in I, and for each cluster 'c' in the propagation tree, we will randomly assign k edges from 'u' with 'c'.
  – Let all those vertices form a set S. Then for some constant number of iterations and while 'u' is non-core (degree of 'u' < *Minpts*), for each vertex 'v' in S, we will check distance between 'u' and 'v'.

- If distance is <= ϵ, edge is created between 'u' and 'v' and degree of 'u' and 'v' both are increased. Then we insert all the neighbours of 'v' in S.
- Since we only require some number of points *(Minpts)* within ϵ distance from 'u' to classify it as core, if this vertex is actually a core vertex after addition of it to the data set, then after some number of iterations it will eventually find some vertices of that cluster in the above process that are within ϵ distance from it.

Efficient computation of $UpdSeed_{Ins}$:
- While running the main algorithm we store the count of number of neighbours of every point.
- Let denote newly identified core node as q'.
- Then, q' will be change into core node if q' $\in N_\epsilon$(p) and initial number of neighbours of q' is *MinPts - 1*.
- Now we need to discover neighbourhood of q' for all q which are core nodes.
- As N_ϵ(p) is already present in the memory, we search for neighbour of q' first there, then only we query neighbourhood of q' and perform an additional region query only if there are more objects in the neighborhood of q' than already contained in N_ϵ(p).

- **Deletion of a point:** Denote the point to be deleted as 'p'. While deleting 'p' from our existing dataset, some of the points can have their degree changed to *Minpts-1* due to their loss of connection with 'p'. So we will store all those core points in I which were in the ϵ-neighbourhood of 'p' since they can become non-core and we could gather some more information regarding these points whether they can become core again or not by checking their neighbourhood.
  Let $Affected_D$(p) is the region affected by the deletion of point.

  We generate a set $UpdSeed_{Del}$ which contains all core node in ϵ-neighbourhood of newly identified non-core nodes. (As deleting point p can make some points non-core from core).

  Mathematically: $UpdSeed_{Del}$ = {q | q is a core object in D\{p}, $\exists$ q': q' is core object in D but not in D\{p} and q $\in N_\epsilon$(q')}

  Efficient computation of N_ϵ(p): We already have ϵ-neighbourhood of p as p was present in the data set before deletion.

  Efficient computation of $UpdSeed_{Del}$:
  - While running the main algorithm we store the count of number of neighbours of every point.
  - Let denote core node in original clustering as q'.
  - Then, q' will be change into non core node if q' $\in N_\epsilon$(p) and initial number of neighbours of q' is *MinPts*.
  - Now we need to discover neighbourhood of q' for all q which are core nodes.
  - As N_ϵ(p) is already present in the memory, we search for neighbour of q' first there, then only we query neighbourhood of q' and perform an additional region query only if there are more objects in the neighborhood of q' than already contained in N_ϵ(p).

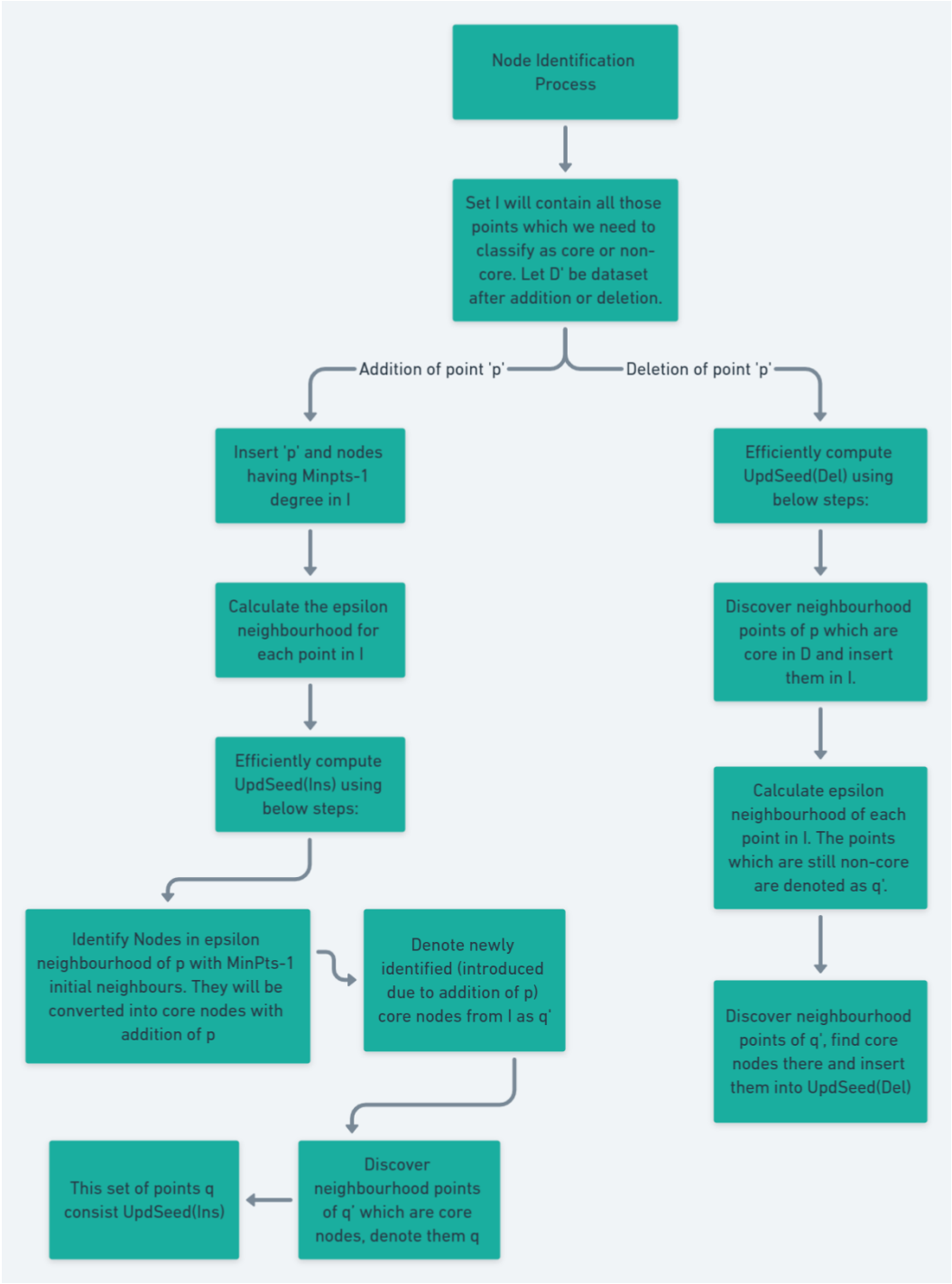

Fig. 13. UpdSeed Construction

(2) **Cluster membership assignment:** Here we perform the merging, seperation of clusters, formation of new clusters due to addition and deletion of points.

- **Addition of a point:** When inserting a new object p, new density-connections may be established, but none are removed. In this case, it is sufficient to restrict the application of the clustering procedure to the set $\text{UpdSeed}_{\text{Ins}}$. If we have to change cluster membership for an object from C to D we perform the same change of cluster membership for all other objects in C. Changing cluster membership of these objects does not involve the application of the clustering algorithm but can be handled by simply storing the information about which clusters have been merged. When inserting an object p into the database D, we can distinguish the following cases:

(a) **Noise:** $\text{UpdSeed}_{\text{Ins}}$ is empty, i.e. there are no "new" core objects after insertion of p. Then, p is a noise object and nothing else is changed.
(b) **Creation:** $\text{UpdSeed}_{\text{Ins}}$ contains only core objects which did not belong to a cluster before the insertion of p, i.e. they were noise objects or equal to p, and a new cluster containing these noise objects as well as p is created.
(c) **Absorption:** $\text{UpdSeed}_{\text{Ins}}$ contains core objects which were members of exactly one cluster C before the insertion. The object p and possibly some noise objects are absorbed into cluster C.
(d) **Merge:** $\text{UpdSeed}_{\text{Ins}}$ contains core objects which were members of several clusters before the insertion. All these clusters and the object p are merged into one cluster.

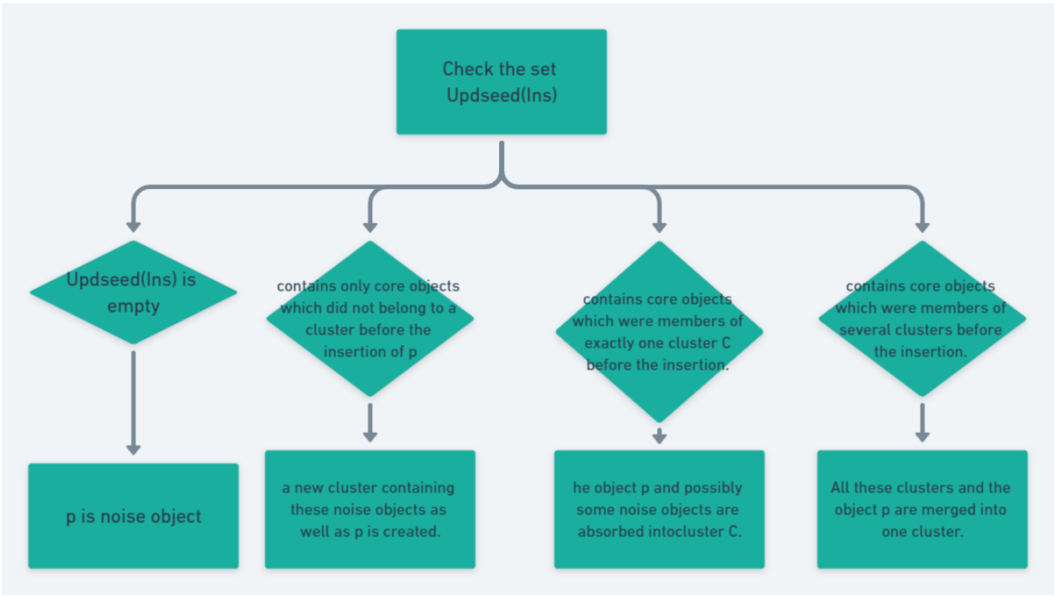

Fig. 14. Insertion of point

- **Deletion of a point:** As opposed to an insertion, when deleting an object p, density-connections may be removed, but no new connections are established. The difficult case

for deletion occurs when the cluster C of p is no longer density-connected via (previous) core objects in $N_\epsilon(p)$ after deleting p. In this case, we do not know in general how many objects we have to check before it can be determined whether C has to be split or not. In most cases, however, this set of objects is very small because the split of a cluster is not very frequent and in general a non-split situation will be detected in a small neighborhood of the deleted object p. When deleting an object p from the database D we can distinguish the following cases:

(a) **Removal:** $UpdSeed_{Del}$ is empty, i.e. there are no core objects in the neighborhood of objects that may have lost their core object property after the deletion of p. Then p is deleted from D and eventually other objects in $N_\epsilon(p)$ change from a former cluster C to noise. If this happens, the cluster C is completely removed because then C cannot have core objects outside of $N_\epsilon(p)$.
(b) **Reduction:** All objects in $UpdSeed_{Del}$ are directly density-reachable from each other. Then p is deleted from D and some objects in $N_\epsilon(p)$ may become noise.
(c) **Potential Split:** The objects in $UpdSeed_{Del}$ are not directly density-reachable from each other. These objects belonged to exactly one cluster C before the deletion of p. Now we have to check whether or not these objects are density-connected by other objects in the former cluster C. Depending on the existence of such density-connections, we can distinguish a split and a non-split situation.

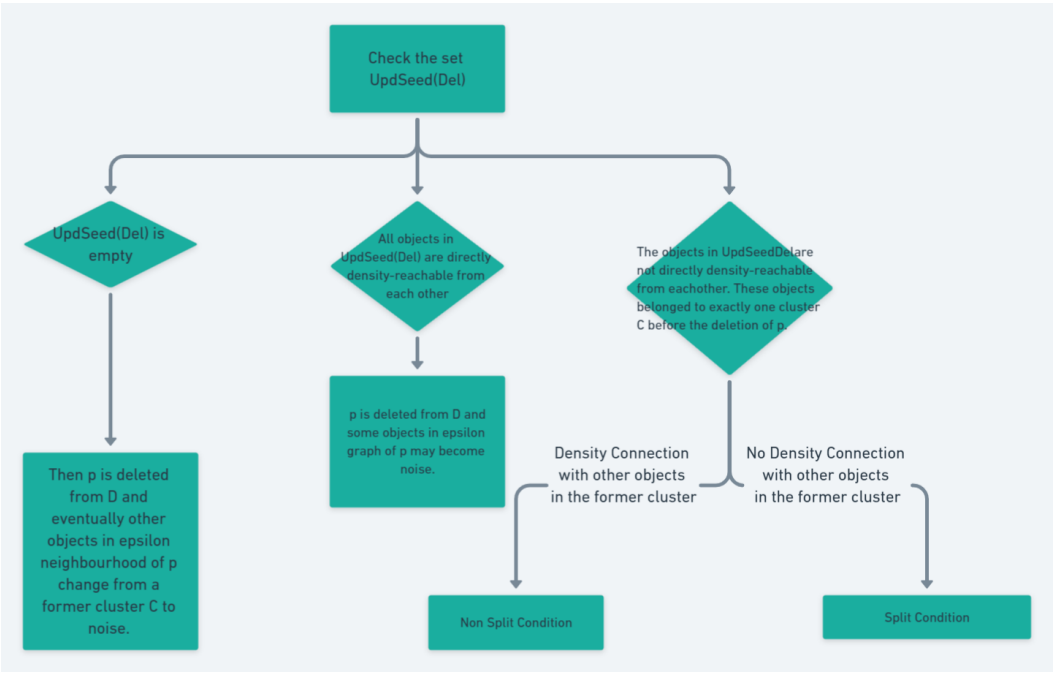

Fig. 15. Deletion of point

INTUTION OF THE PROPOSED APPROACH

Due to the density-based nature of NG-DBSCAN, the insertion or deletion of an object affects the current clustering only in the neighborhood of this object.

(1) **Intution for node identification:** To classify each node 'p' as core node, we only need to have *Minpts* number of neighbours within ϵ distance from 'p'. Hence to gather these *Minpts* neighbours for all points in I, then if in reality there are *Minpts* neighbours for current point in the dataset then we will be able to find such set of points in one or more clusters after some iterations.
Here we are reducing our computations of finding the ϵ-neighbourhood of whole graph again to only finding it for the points in I.

(2) **Intution for finding cluster membership:** We have proved that only for points inside the $UpdSeed_{Ins}$ and $UpdSeed_{Del}$ are the nodes for the update related to changing of cluster memberships.
Here we are reducing our computations by looking over only certain nodes while performing addition and deletion of points instead of looking over and creating the whole propagation tree and clusters again.

FURTHER WORK

In next week we will work on detailed algorithm of Cluster Membership Assignment for insertion and deletion both. We will further compare the percent of time needed to run the whole NG-DBSCAN algorithm with respect to insertion of new coming data objects and determines which approach takes fewer amounts of time and effort.