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.

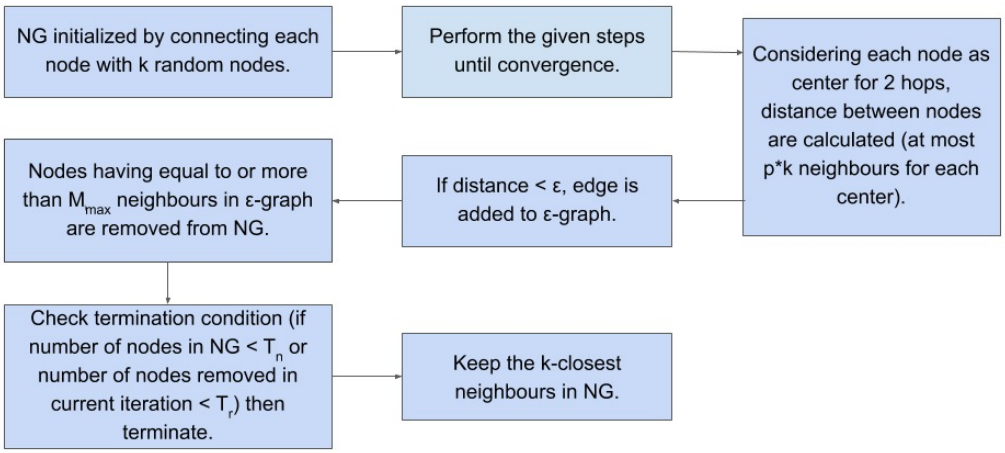

Fig. 1. Overview of Phase 1

(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

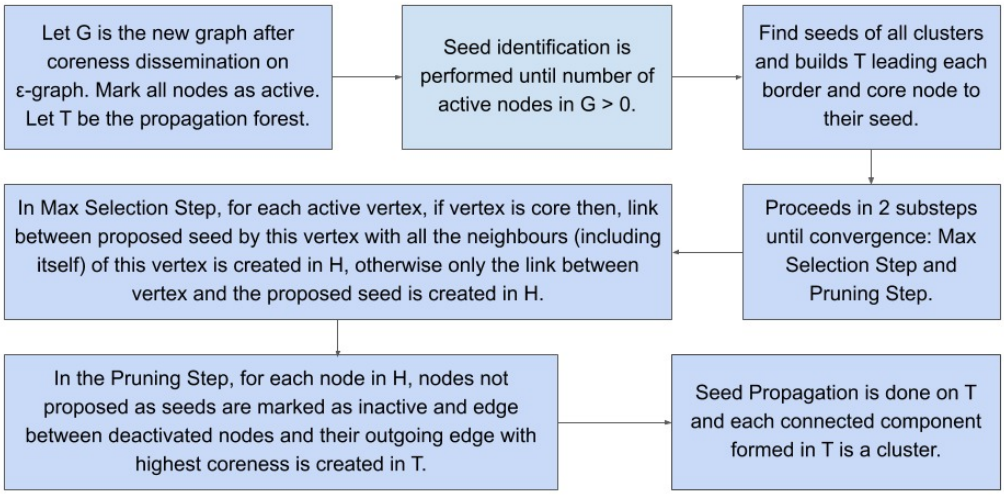

Fig. 2. Overview of Phase 2

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()
   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()
   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)
    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.