An Incremental Structural Clustering Algorithm for Networks
Group 8: The Hitchikers

Annanya Pratap Singh Chauhan (170101008)
Arpit Gupta (170101012)
Aryan Agrawal (170101013)
Shivang Dalal (170101086)
Github Repository: https://github.com/SDTheSlayer/data-mining

ABSTRACT

Graph Clustering is an essential component in mining information from a network. It reveals hidden structures in a network. Structural Clustering Algorithm for networks (SCAN) is a pivotal work in the world of graph clustering. The SCAN algorithm can not only identify clusters but can also identify hubs and outliers. SCAN also acts as a basis for a broad range of clustering algorithms. It works well for static graphs, but most modern real-world graphs are dynamic; hence it is very inefficient for dynamic graphs. Through this project, we propose an efficient incremental version of SCAN for dynamic networks, called Incremental SCAN. Incremental SCAN can save us a lot of computation effort during incremental updates on the graph by avoiding cluster computation from scratch.

1 INTRODUCTION

A wide range of the data used in scientific research can be represented as graphs. A graph is a set of vertices representing objects connected by edges; these edges represent the relationship between the objects. Many real-life scenarios can be modeled in the form of graphs, such as Social media networks where edges represent the relationship between users. The worldwide web can also be modeled as a graph where vertices represent web pages, and edges represent hyperlinks.

Most of these real-world graphs carry community clusters within them. Discovering these clusters can provide us with valuable insights. A cluster of people identified in a social media network is expected to share similar interests. Similarly, in a biological graph, a cluster of molecules can share similar properties. The graphs might also contain hubs, which are not part of any cluster but act as an interconnect between two different clusters. In an epidemiological study, hubs are responsible for the rapid spread of disease. Then there are outlier nodes that are neither part of any cluster nor connect two different clusters. Outliers can often be ignored in our real-world analysis.

Graph clustering is a vital method in identifying these structures. In the past two decades, there have been many proposed algorithms and models for graph clustering. The most prominent among these is SCAN ([16]). Compared with its compatriots, one of the most attractive features of SCAN is its ability to identify hubs and outliers. Vertices are grouped based on the neighbors they share. SCAN uses the neighborhood similarity of vertices as a clustering criterion instead of direct edges. The neighborhood similarity of two vertices is quantified using the structural similarity value for their edge. Only edges with structural similarity more significant than epsilon are considered. The algorithm then identifies core vertices(vertices with more than μ neighbors) and starts creating clusters from these cores in a BFS fashion. After the clustering is completed, the remaining vertices are classified as either hubs or outliers. More details about SCAN and its variants are discussed in Related Works (3). SCAN was initially intended for static graphs. However, modern systems ask us for solutions that can work in an incrementally distributed manner. So we propose an Incremental SCAN that can:

- Update clusters, hubs, and outliers on edge addition/deletion.
- Update clusters, hubs, and outliers on vertex addition/deletion.
- Parallelize edge similarity calculation across multiple worker nodes.

Incremental SCAN works by avoiding recomputation of all the SCAN-generated clusters. On an edge addition or deletion, the structural similarities of only a subset of edges are affected. The reduction of structural similarity of an edge might lead to splitting of clusters, while an increase in structural similarity of an edge might lead to the merging of two clusters. More details regarding the Incremental approach are discussed in section 4. We performed extensive analysis to prove that the proposed Incremental SCAN has a better running time than the original SCAN. We also analyze the advantages/disadvantages of multithreaded similarity calculation in Incremental SCAN and SCAN.

The rest of the paper is organized as: Section 2 gives a brief introduction of SCAN, Section 3 gives a review of related works. Section 4 demonstrates the proposed Incremental algorithm. Results and their analysis is given in Section 5. Our learnings from this project and future scope of the project has been outlined in sections 6 and 7, respectively.

2 REVIEW OF SCAN [16]

SCAN is one of the first algorithms that uses the neighborhood of vertices as a clustering criterion instead of direct edges. Most of the previous graph clustering methods aimed to maximize the number of edges within a cluster and minimize the number of edges between clusters. Some of these algorithms are Modularity based algorithms [6] [12] [4] and Normalized cut algorithms [5] [10]. However, these algorithms do not differentiate between the purpose of vertices in the graph. Some vertices are hubs, which act as a bridge between clusters. Others are outliers, which do not have any substantial relation to any cluster and are neither hubs.

SCAN improved upon the existing algorithms as:

- It detects hubs and outliers in addition to just clustering.
- It is fast. With n vertices and m edges, it has a running time of $\mathcal{O}(m^{1.5})$. In comparison, the fastest existing clustering algorithm (fast modularity based algorithm [4]) is $\mathcal{O}(md \log n)$.

Authors used two parameters, run time complexity and ARI to compare SCAN with FastModularity algorithm[4] and obtained better results in both. SCAN has been implemented in CDlib library[13], a Python language software package for the extraction, comparison,and evaluation of communities from complex networks. However, no C++ implementation could be found for SCAN.

2.1 Terminology

We consider an undirected and unweighted graph $G(V, E)$ where V is the set of vertices and E is the set of edges. We denote the number of vertices $|V|$ and the number of edges $|E|$ by n and m respectively. Following notations have been used to define the algorithm:

- **VERTEX STRUCTURE** For $v \in V$, vertices in its neighbourhood including itself is called its vertex structure. It is denoted by $\tau(v)$.
- **STRUCTURAL SIMILARITY** For any two vertices $v, w \in V$, structural similarity $\sigma(v, w)$ is defined as:

$$\sigma(v, w) = \frac{\|\tau(v) \cap \tau(w)\|}{\sqrt{\|\tau(v)\|\|\tau(w)\|}} \tag{1}$$

- **ϵ-NEIGHBOURHOOD** For a vertex $v \in V$, ϵ-neighbourhood is defined as the set of vertices in its vertex structure that have the structural similarity at-least $\epsilon \in \mathbf{R}$. Formally,

$$N_\epsilon(v) = \{w \epsilon \tau(v) \| \sigma(v, w) \geq \epsilon\} \tag{2}$$

- **CORE** A vertex $v \in V$ is called a core, if for given $\epsilon \in \mathbf{R}$ and $\mu \in \mathbf{N}$ number of vertices in ϵ-neighbourhood of v should be atleast μ.

$$CORE_{\epsilon,\mu}(v) \iff \|N_\epsilon(v)\| \geq \mu \tag{3}$$

- **DIRECT STRUCTURE REACHABILITY** All the vertices in the ϵ-neighbourhood of the core are said to be directly reachable from core.

$$DirREACH_{\epsilon,\mu}(v, w) \iff CORE_{\epsilon,\mu}(v) \wedge w \in N_\epsilon(v) \tag{4}$$

- **STRUCTURE REACHABILITY** A vertex $w \in V$ is structure reachable from $v \in V$ w.r.t ϵ and μ , if there is a chain of vertices $v_1, ..., v_n \in V, v_1 = v, v_n = w$ such that v_{i+1} is directly structure reachable from v_i.
- **STRUCTURE CONNECTIVITY** A vertex $v \in V$ is structure-connected to a vertex $w \in V$ w.r.t ϵ and μ , if there is a vertex $u \in V$ such that both v and w are structure reachable from u.
- **STRUCTURE-CONNECTED CLUSTER** For given $\epsilon and \mu$, a non-empty subset $C \subseteq V$ is called as structure connected cluster if all vertices in C are structure connected and it is maximal w.r.t to structural reachability.
- **CLUSTERING** For given graph G, set of all the structure-connected clusters is clustering.
- **HUB** if a vertex $v \in V$ does not belong to any structure connected clusters and has neighbors belonging to two or more different clusters then it is a hub. In simpler words it bridges two or more clusters.
- **OUTLIER** if a vertex $v \in V$ does not belong to any structure connected clusters then it is called an outlier if and only if all its neighbors either belong to only one cluster or do not belong to any cluster.

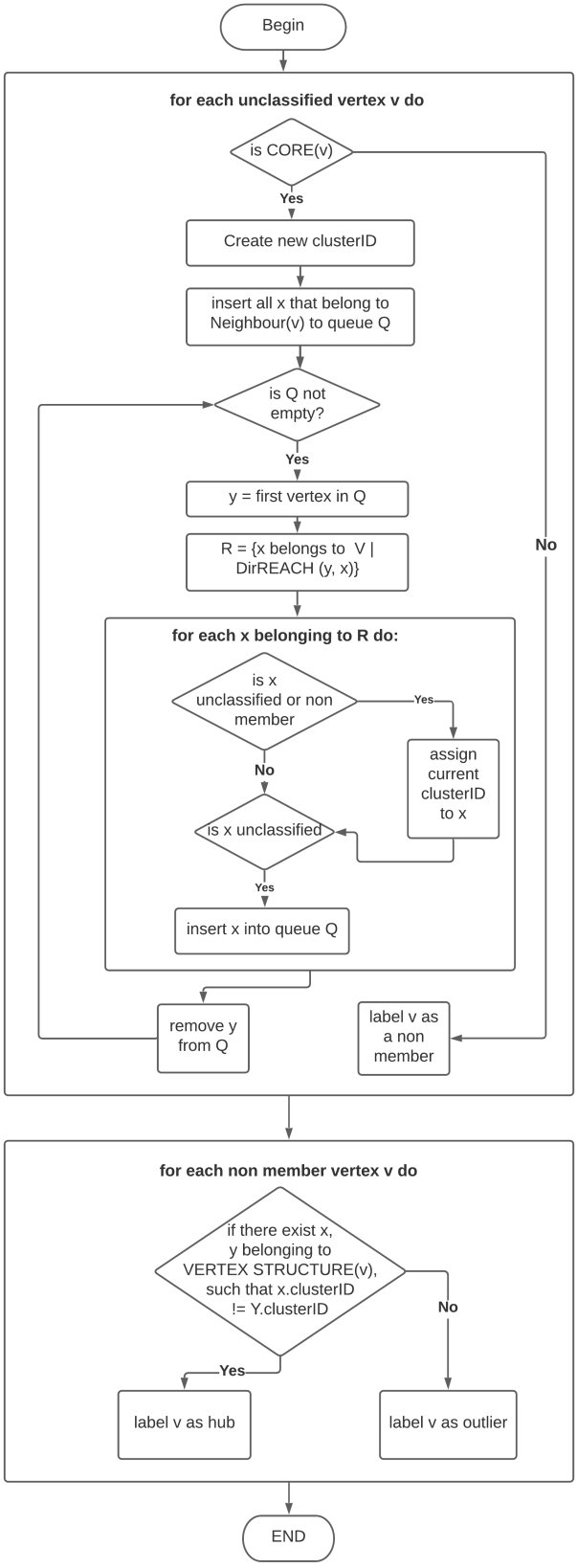

Figure 1: Flowchart of SCAN

2.2 Algorithm

SCAN algorithm is based on the idea of identifying the core, hubs, and outliers. Firstly all the vertices are marked as unclassified (.i.e., they do not belong to any clusters). Then iterating over all the vertices, it is checked if they are core are not. If the vertex is found out to be core, then a new cluster is formed. Otherwise, the vertex is labeled as a non-member. To find a new cluster from the core vertex, a new clusterId is generated, and all the directly structure reachable vertices from the core are inserted in a queue. A breadth-first search is then performed to find all the structure reachable nodes from the core and assign the current ClusterId.

After forming the clusters, there could be some vertices that do not belong to any clusters (non-member vertices). Iterate over these nodes to mark them as hub or outlier as per their definition.

3 RELATED WORKS

In this section we will briefly describe the various modifications attempted over SCAN over the years and other related works.

3.1 SCAN++ [14]

This algorithm addresses the issue of the high time complexity of SCAN on large graphs. It exploits a property of real-world graphs that they have high scores of clustering coefficients (node density), i.e., a node and its two-hop-away node in real-world graphs are expected to share large parts of their neighborhoods. It introduces a new data structure, directly two-hop-away reachable node-set (DTAR), the set of nodes that are two hops away from a given node. It drops unnecessary density evaluations for adjacent nodes in the clustering procedure by using DTAR.

3.2 AnySCAN [17]

It uses the active learning strategy to find the same clustering result on large-scale networks as the original SCAN in a significantly more efficient manner. It applies anytime algorithm to the network clustering problem. Anytime algorithms' basic procedure is deriving an initial solution quickly to a time-consuming problem and then refining iteratively until the available time runs out, or the solution is acceptable. It proposes an active learning strategy to refine clustering results iteratively. In particular, large-scale networks' vertex structure is exploited to choose more informative vertices to boost refining procedure efficiency.

3.3 pSCAN [2]

This algorithm mainly tries to solve two main challenges for SCAN, reduce the number of structural similarity calculations and efficiently check if two vertices are structurally similar.

Firstly this paper proves that SCAN is worst-case optimal but still is not scalable for large-scale graphs due to computation of structural similarity for each vertices pair.

Secondly, this paper makes three observations about structural graph clustering:

- The clusters may overlap
- The clusters of core vertices are disjoint
- The clusters of non-core vertices are uniquely determined by core vertices

Based on these observations, the authors develop a two-step paradigm for clustering. They first cluster core vertices and then cluster non-core vertices by assigning them the same cluster as their neighbors, which are core and structurally similar.

Thirdly the authors propose three optimization techniques, cross-link, pruning rules, and adaptive structure-similar checking, for speeding up the checking of structure-similar between two vertices without computing the exact value of their structural similarity.

3.4 Efficient Structural Graph Clustering: An Index-Based Approach [15]

This paper tries to solve the following challenges, parameter tuning (μ and ϵ), number of structural similarity calculations, and network update. Authors propose a novel index structure GS*-Index. It has two main parts core-orders and neighbor-orders. Based on GS* index clusters are computed in $O(\sum_{C \in \gamma} |\mathrm{E}_C|)$ time where γ is the result set of all clusters and $|\mathrm{E}_C|$ is the number of edges in specific cluster γ. The authors also provide algorithms to update GS* index on edge addition and deletion.

3.5 Incremental Structural Clustering for Dynamic Networks [3]

ISCAN tries to convert the original SCAN algorithm into an incremental version with minimal additions or modifications. Their key observation is that the SCAN algorithm while clustering, creates a BFS Forest with each tree being a cluster. Hence this forest can be stored while running the SCAN algorithm and then incrementally modified. All non-leaf nodes of the trees are core vertices. They store all edges that lie above the similarity threshold but do not belong to any tree as a separate set.

The authors argue that when an edge is updated, i.e., inserted or deleted, between the nodes a and b, only the similarity of edges having one vertex in the union of the neighborhoods of a and b $[N_\epsilon(a) \cup N_\epsilon(b)]$ and other in $\{a, b\}$ need to be updated. This ensures that only a limited number of edges need to be checked which drastically reduces the computation complexity. Once these edges' similarities are updated, the clusters can be modified by adding or removing these edges from the BFS trees.

An in depth analysis of ISCAN paper revealed several key steps missing in its algorithm and an incorrect computational complexity. The major part which was glossed over by the authors while calculating the computational complexity of ISCAN was the merging of clusters, i.e combining of clusters due to increase in its similarity value. They simply counted it as the addition of an edge ($O(1)$) operation, however the cluster ids of the merged cluster needs to be updated. So for the entire loop they state a worst case computation complexity of $O(n)$ whereas it is actually $O(nlogn)$. The second flaw is in the proposed condition for rejoining clusters after splitting. Their condition for rejoining is fundamentally wrong, since it will lead to incorrect clustering. These and other omissions have motivated us to modify the ISCAN algorithm.

4 PROPOSED INCREMENTAL ALGORITHM

This section outlines the proposal for an incremental variant of SCAN. We build upon the critical observations put forth by ISCAN[3]. First, we shall describe these observations and the modifications to

Algorithm 1: SCAN

Input : $G = (V, E)$, ϵ and μ
Output : The BFS-forest and the non-tree-edge set ϕ

calculateAllSimilarity();
forall *Unclassified vertex* $v \in V$ **do**
  if v *is CORE* **then**
    Generate new clusterID for v;
    forall $x \in N_\epsilon(v)$ **do**
      Insert x into queue Q;
      $x.parent = v$;
    end
    while $Q \neq \emptyset$ **do**
      $y = Q.pop()$;
      $R = \{x \in V | DirREACH_{\epsilon,\mu}(y, x)\}$;
      forall $x \in R$ **do**
        if x *is* $unclassified$ **then**
          Assign the current clusterID to x;
          Insert x into queue Q;
          $x.parent = y$;
        else
          if $x.parent \neq y \wedge y.parent \neq x$ **then**
            Insert the edge (y, x) into ϕ;
      end
    end
end
forall $NON - MEMBER$ *vertex* v **do**
  if $(\exists\, x, y \in \Gamma(v) : x.clusterID \neq y.clusterID)$ **then**
    Label v as hub;
  else
    Label v as outlier;
end

Algorithm 2: Update Edge

Input : $G = (V, E)$, ϵ and μ, the updated edge $e = (u, v)$, the non-tree-edge set ϕ
Output : The updated clusters

$\mathbb{N}(e_{uv}) = \tau(u) \cup \tau(v)$
$\mathbb{R}(e_{uv}) \subseteq E$ where one end-point belongs to $\mathbb{N}(e_{uv})$ and other belongs to $\{u, v\}$
Recompute the structural similarity for all edges in $\mathbb{R}(e_{uv})$
forall $w \in \mathbb{N}(e_{uv})$ **do**
  if w *is CORE* **then**
    MergeCluster(w, ϕ);
end
forall $e = (u, v) \in \mathbb{R}(e_{uv})$ **do**
  if $\sigma_{old} \geq \epsilon \wedge \sigma_{new} < \epsilon$ **then**
    SplitCluster(e, ϕ);
end
Run BFS on BFS forest and recompute clusters;
forall $e = (u, v) \in \phi$ **do**
  if $u.ClusterID \neq v.ClusterID$ **then**
    if u *is CORE* $\wedge$ v *is CORE* **then**
      Merge(u, v);
    if u *is CORE* $\wedge$ v *is* $NON - MEMBER$ **then**
      $v.ClusterID = u.ClusterID$;
      Create edge (u, v) in BFS-tree;
      Remove (u, v) from ϕ;
    if v *is CORE* $\wedge$ u *is* $NON - MEMBER$ **then**
      $u.ClusterID = v.ClusterID$;
      Create edge (u, v) in BFS-tree;
      Remove (u, v) from ϕ;
end
forall $NON - MEMBER$ *vertex* v **do**
  if $(\exists\, x, y \in \Gamma(v) : x.clusterID \neq y.clusterID)$ **then**
    Label v as hub;
  else
    Label v as outlier;
end

the SCAN algorithm needed. The first insight is that when an edge (u,v) is updated, the similarity of only a limited number of edges is changed. This can be quantified as the set of vertices $\mathbb{R}(e_{uv}) \subseteq E$ where one end-point belongs to $\mathbb{N}(e_{uv})$ and the other belongs to $\{u, v\}$, where $\mathbb{N}(e_{uv})$ is the union of the neighborhoods of u and v. On adding or removing the edge (u,v), only the similarity of edges in $\mathbb{R}(e_{uv})$ needs to be updated. The similarity of any edge (w,v) or (w,u) will increase if w is connected to both u and v. And and if w is only connected to one of u or v, then the similarity of that edge will decrease. An analysis of deletion will lead to similar results, with reverse change in similarity values.

Secondly, the SCAN algorithm can be modified to store the BFS path followed during clustering in the form of a tree. All non-tree edges that correspond to directly reachable edges are stored in a separate set called ϕ. This set is needed since it may be possible to stop a cluster from splitting due to an edge removal by adding an edge from the ϕ set into the bfs tree. A flow chart describing the algorithm is depicted in figure 2

4.1 Algorithm Description

We first run the modified version of SCAN on the available dataset as outlined in 5.1. Then on each edge updation, the edge update algorithm is called as described in algorithm 2. First, the structural similarity of all the edges in $\mathbb{R}(e_{uv})$ is recalculated. This is argued to have a computation complexity of $O(m)$ for real-world graphs since they are sparse and $|\mathbb{R}(e_{uv})|$ is generally very small as discussed by the authors of ISCAN [3]. Next for each core vertex w in $\mathbb{N}(e_{uv})$ the MergeCluster [3]function is called.

The MergeCluster function firstly generates a new clusterID in case w is unclassified and assigns it to w. Next, it iterates over all vertices in the epsilon neighborhood of w. For each such vertex, u if it is unclassified, it directly assigns the clusterID of w to u and makes w the parent of u. In case u is classified, then it checks if u is a non-core vertex. In the non-core case, if it has a clusterID to ϕ. The second case is when u is a core vertex. In this scenario, if both u and w are in the same cluster and neither is the parent of the other, then the edge is again added to ϕ. The last case is when u is a core but belongs to a different cluster, this implies that the two clusters need to be merged. This merging process is done by running a

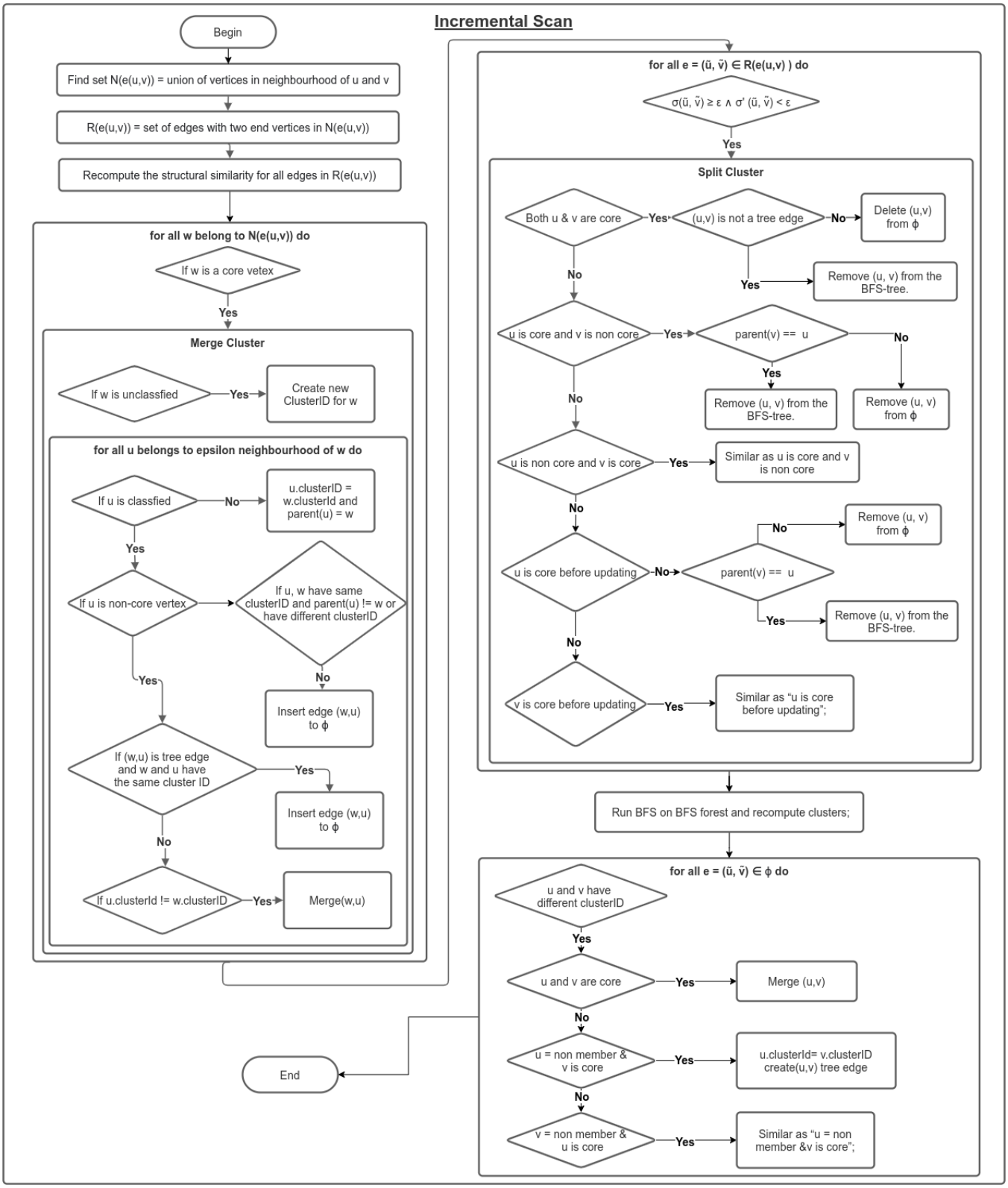

Figure 2: Incremental SCAN

Algorithm 3: MergeCluster(w, ϕ)

```
if w is unclassified then
    label w as core;
    Create a new clusterID for w;
forall u ∈ N_ε(w) do
    if u is classified then
        if u is NON-CORE vertex then
            if (u.ClusterID = w.ClusterID ∧ u.parent ≠ w) ∨ (u.ClusterID ≠ w.ClusterID) then
                Insert edge(w, u) into φ;
        else
            if u.ClusterID = w.ClusterID ∧ u.parent ≠ w ∧ w.parent ≠ u then
                Insert edge (w, u) into φ;
            if u.ClusterID ≠ w.ClusterID then
                Merge(w, u);
    else
        u.ClusterID = w.ClusterID;
        u.parent = w;
        Insert (u, w) in BFS-tree;
end
```

Algorithm 4: Merge(u, v)

```
if SizeOfCluster[u.ClusterID] >= SizeOfCluster[v.ClusterID] then
    Remove (u, v) from φ;
    Add (u, v) to the BFS-tree;
    v.parent = u;
    Run BFS on the subtree of v starting with v, and setting clusterID of all nodes in the subtree as u.ClusterID, also adjust parent-children relations in such a way to make v the root of its subtree;
else
    Remove (u, v) from φ;
    Add (u, v) to the BFS-tree;
    u.parent = v;
    Run BFS on the subtree of u starting with u, and setting clusterID of all nodes in the subtree as v.ClusterID, also adjust parent-children relations in such a way to make u the root of its subtree;
```

BFS on all the tree edges of the smaller cluster (starting from u or w, depending on which is the smaller cluster) and updating the clusterIDs and parents. The MergeCluster algorithm over all the vertices has a worst-case run time complexity of $O(nlogn)$ in the case where you go from n single vertex clusters to one cluster.

The above conditions took care of all the cases where the structural similarity of an edge may increase. Next, we handle the cases where it may decrease. For all edges in $\mathbb{R}(e_{uv})$, we check if their similarity was previously above ϵ but is now lower than ϵ. This means that the edge needs to be removed from either the tree edge set or the ϕ set, depending on its current location. The SplitCluster function takes care of this and has been outlined in the form of

Algorithm 5: SplitCluster($e = (u, v), \phi$)

```
if u is CORE ∧ v is CORE then
    if u.parent ≠ v ∧ u ≠ v.parent then
        Remove (u, v) from φ;
    else
        Remove (u, v) from the BFS-tree;
if u is CORE ∧ v is not CORE then
    if v.parent = u then
        Remove (u, v) from the BFS-tree;
    else
        Remove (u, v) from φ;
if v is CORE ∧ u is not CORE then
    if u.parent = v then
        Remove (u, v) from the BFS-tree;
    else
        Remove (u, v) from φ;
if u is NOT − CORE ∧ v is NOT − CORE then
    if u is CORE before updating then
        if v.parent = u then
            Remove (u, v) from the BFS-tree;
        else
            Remove (u, v) from φ;
    if v is CORE before updating then
        if u.parent = v then
            Remove (u, v) from the BFS-tree;
        else
            Remove (u, v) from φ;
```

an algorithm [5]. After removing all the edges from the tree set, we run a BFS on the trees to form the new clusters after splitting. Finally, we try and merge some of these clusters again by using the edges from ϕ. For this, we iterate over all edges (u,v) in ϕ, and if the two vertices belong to different clusters and are both core vertices, we merge the two clusters as described previously. If one of the vertices u is non-member, then it is assigned the clusterID of v, and v is made the parent of u.

The above computation ensures optimal clustering of all vertices. Lastly, we need to classify the non-member vertices into hubs and outliers. For this, we iterate over all vertices, identify the non-member vertices and utilize the same conditions as the original SCAN to classify them into hubs or outliers. The time complexity for similarity calculation is taken as $O(m)$ as reasoned above, the merge algorithm has a time complexity of $O(nlogn)$, hence the merge cluster algorithm runs in $O(m + nlogn)$. The split cluster algorithm takes a total time of $O(m)$, and further BFS runs and re clustering again takes $O(m+nlogn)$. Hence the final time complexity of the incremental algorithm for each edge updation is $O(m+nlogn)$.

4.2 Incremental SCAN Code Architecture

Now we describe the code architecture used for implementing the incremental variant of SCAN. Figure: 3 depicts the class diagram. The description of the various classes designed is as follows:

(1) **Vertex**: This class represents a vertex of the graph. It stores the vertex ID, Name and $ClusterId$. The algorithm uses $IsClassified$

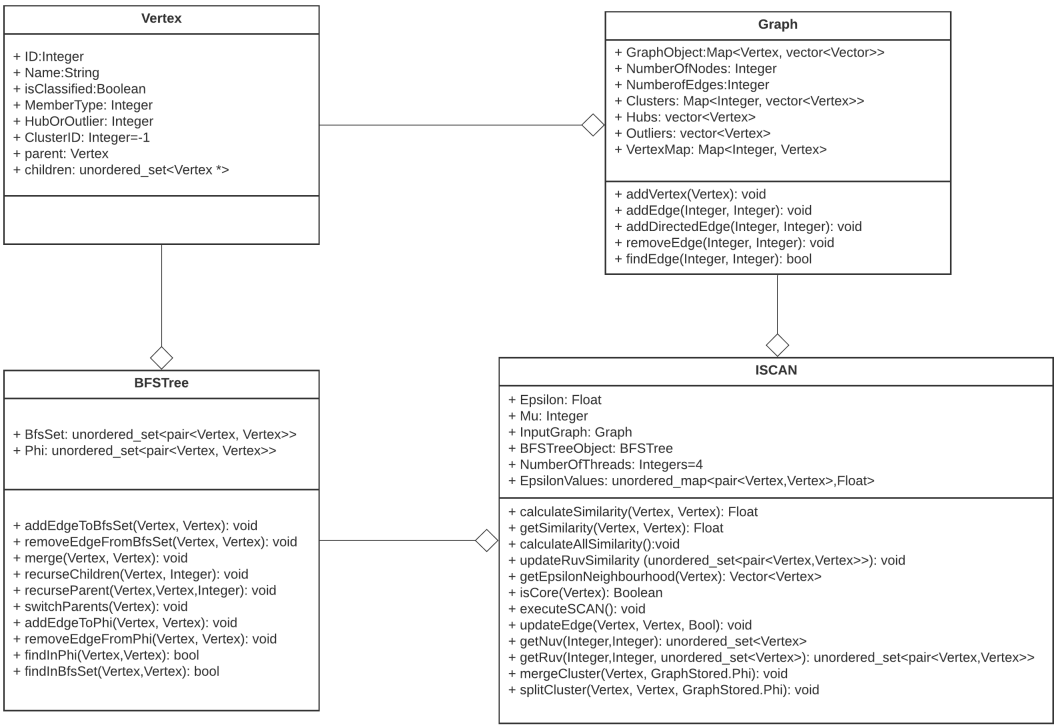

Figure 3: Incremental SCAN Code Architecture

to mark the vertices it has analyzed. *Memeber Type* is used to store whether the vertex is a Core or *NonCore Member* or *NonMember* (Hub/Outlier). *HubOrOutlier* is used to classify *NonMember* vertices into either hubs or outliers. *Parent* and *children* are the parent and children of the vertex in the bfs tree.

(2) **Graph**: Graph stores the vertices and edge set for the graph as well as the results of the clustering algorithm. It has a hash map that stores the edges in the form of an adjacency list. The SCAN algorithm results are stored in the form of vertex clusters, a set of hubs, and a set of outliers. $VertexMap$ is used for mapping vertex id and name to its object pointer. The total number of edges and vertices are also stored separately in the class. The Graph class also has some essential functions like *addEdge* and $addVertex$ for creating the graph.

(3) **BFSTree**: The $BFSTree$ object is used to store all the edges belonging to the bfs and phi set in the $BfsSet$ object and *Phi* object respectively. Both use an unordred set of vertex pairs to store the edges. It has basic functions to add, remove and find edges in both the edge sets. It also contains the *merge* function, which merges a BFS tree belonging to a different cluster into the current BFS tree. The *recurseParent* and *recurseChildren* are helper functions for the merge function which help in traversing the target bfs tree.

(4) **ISCAN**: This is the main class for the incremental version of the SCAN algorithm. It has a primary function *executeSCAN* for the initial clustering of vertices and similarity calculation by *calculateAllSimilarity*. Furthermore, an *updateEdge* is called each time a new edge is added or removed from the graph to recalculate the structural similarity of required vertex pairs and merge or split clusters as per the change in structural similarity. Some other important functions are *mergeClusters* and *splitClusters* to merge and split clusters as defined in the proposed algorithm. The ISCAN object also stores the input parameters μ, ϵ, and $NumberofThreads$, the input graph and a $BfsTreeObject$ to represent clusters. There are some helper functions like *getSimilarity*, *getEpsilonNeighbourhood*, and *isCore* to get the similarity, get the epsilon neighbouhood and check if a vertex is a core, respectively. The similarity values for edges are stored in a set $EpsilonValues$, and is updated for the R_{uv} set for each edge updation using *updateRuvSimilarity*. Other helper functions include *getRuv* and *getNuv* to return the respective sets.

We have used various data structures in our implementation. We have used an STL hash map from vertex to a vector of vertices to represent our graph. This is an adjacency list representation of the graph. To represent our Clusters, we have created a hash map from *clusterID* to a vector of vertices, where the vector represents a cluster. We have used vectors of vertices to represent hubs and outliers. $VertexMap$ is an STL hash map from integer, a string to a vertex. Similar structures have been used for BFS class as well.

4.3 Further improvements to incremental SCAN

An option has also been added to add and delete vertices. To accomplish the deletion, we iteratively delete all the vertex edges and then remove the vertex from all the corresponding data structures.

Hence an additional function has been created in the ISCAN class for creating or deleting a vertex. To verify the implementation of the same, we tried only adding vertices into the graph when an edge corresponding to it is added and comparing the clusters with SCAN. Deletion has also been verified by deleting random vertices one by one till the graph is empty. Since this is not available in the original SCAN, there is no baseline for comparisons.

We have also implemented a multithreaded version of the similarity calculation function. We can do this as all similarity value calculations are independent of each other. To implement this, we have added an option to enable multithreading in the ISCAN class, which allows the creation of multiple threads, each being assigned the similarity calculation of specific edges. The multithreading option speeds up the computation of similarity values, specifically when running the original SCAN algorithm.

5 EXPERIMENTAL ANALYSIS

In this section we evaluate the algorithm incremental SCAN with both synthetic datasets and real datasets. We compared its performance with the SCAN algorithm on various cases of addition and deletion of edges and vertices. We implemented both the SCAN and incremental SCAN algorithm in C++.

The system configuration used for all of the results is the following: 8 core 16 thread Intel(R) Xeon(R) CPU E5-2630 v3 @ 2.40GHz, with 126 GB of RAM. The results were run on a shared server when it had next to no load, however a few disparities in the results could be attributed to other processes being run parallelly on the server.

5.1 Datasets

We have used in total of 7 graph dataset as summarized in table 1. We used two synthetic graph dataset: Example with 13 nodes and 27 edges and Customer with 16 nodes and 38 edges. These graphs were taken from the SCAN paper.

We used two small real world graph datasets as used in SCAN for evaluation. The first small real world dataset is football schedule of NCAA Football Bowl Subdivision (formerly Division 1-A), 2006 [1]. It consists of 115 nodes interconnected by 613 edges. The second small real world dataset is the classification of books about US politics compiled by Valdis Krebs [11]. It consists of 105 nodes interconnected by 441 edges.

We have considered three large real datasets after slight preprocessing.First is General Relativity and Quantum Cosmology collaboration network (GR-QC)[9] with 14 thousand edges and 5 thousand nodes. It covers scientific collaborations between authors of papers submitted to General Relativity and Quantum Cosmology category. If an author i co-authored a paper with author j, the graph contains a undirected edge from i to j. If the paper is co-authored by k authors this generates a completely connected (sub)graph on k nodes. Second is Condense Matter collaboration network (COND-MAT)[8] with 93 thousand edges and 23 thousand nodes. It covers scientific collaborations between authors of papers submitted to Condense Matter category. Third is Astro Physics collaboration network (ASTRO-PH) [7] with 2 lakh edges and 19 thousand nodes. It covers scientific collaborations between authors of papers submitted to Astro Physics category.

Table 1: Datasets

Datasets	Number of Nodes	Number of edges
Example	13	27
Customer	16	38
Polbooks	105	441
Football	115	613
GR-QC	5242	14496
COND-MAT	23133	93497
ASTRO-PH	18772	198110

5.2 Evaluation

We ran multiple experiments of edge addition and deletion to compare the single threaded and multi-threaded version of incremental SCAN with the naive approach of rerunning the SCAN algorithm. We have also implemented a correctness checking algorithm which matches the clusters between the two implementations on every addition. We ran the correctness checking algorithm for graphs upto 1000 edges, since this has a run time complexity of $O(n^3)$ and hence not viable on larger graphs. For all the datasets used by the original SCAN paper the incremental version outputs the correct answer for edge deletion and addition. To test this we tried adding all the edges one by one from empty graph to the complete dataset and also removing one by one. Also confirmed by adding and deleting random edges. Hence we have not used accuracy as a metric since for all current tests it has been a 100% accurate.

For evaluation we first read a file containing information of vertices and edges of graph and create a graph object. This graph object , and user input ϵ and μ is then used to create object SCAN and incremental SCAN objects. Then the operations of edge deletion and addition are performed separately on each object. Results of performance analysis is summarized below.

- **Running Time by adding all edges one by one**: We performed this test for all the graphs in the SCAN paper by one by one adding all edges and running the incremental (single threaded and multi-threaded) and normal SCAN at each step. These results can be seen in Figure 6. As we can observe the incremental algorithm performed vastly better than original SCAN. Also single threaded version of incremental algorithm performs better than its multi-threaded version.
- **Running Time by deleting all edges one by one**: We performed this test for all the graphs in the SCAN paper by one by one deleting all edges from the final dataset to empty and running the incremental and normal SCAN at each step. These results can be seen in Figure 7. As we can observe the incremental algorithm performed vastly better than original SCAN. Also single threaded version of incremental algorithm performs better than its multi-threaded version as was in the case of edge addition.
- **Addition of random edges on large datasets:** We then ran the incremental and SCAN on real world large datasets. We added the last 500 edges incrementally onto the existing graph and compared the results. For incremental version we simply called for edge updation method whereas for SCAN we created a new graph including the added edges and then reran the SCAN algorithm on this new graph. Figure 5 depicts

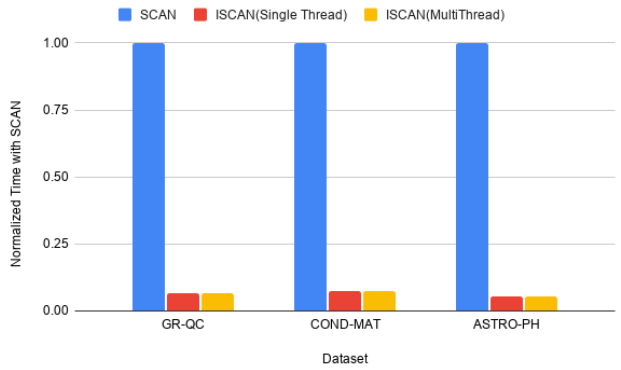

Figure 4: Comparision of run time of deletion by SCAN, Incremental SCAN and Incremental SCAN with multithreading algorithms on large datasets.

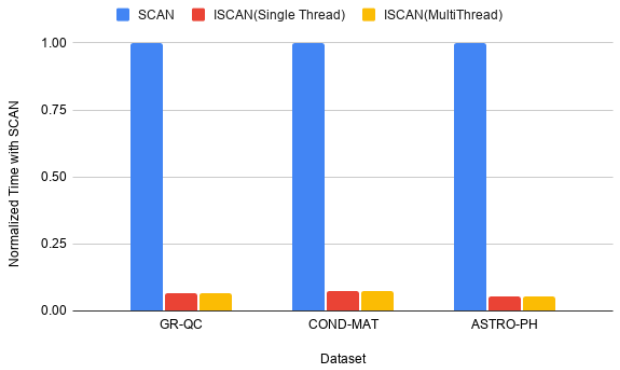

Figure 5: Comparision of run time of addition by SCAN, Incremental SCAN and Incremental SCAN with multithreading algorithms on large datasets.

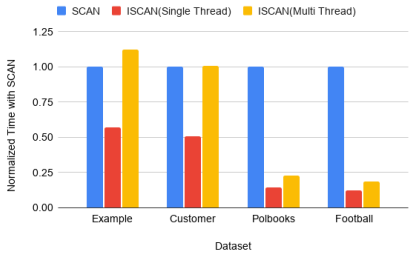

Figure 6: Comparision of run time of addition by SCAN, Incremental SCAN and Incremental SCAN with multithreading algorithms on small datasets.

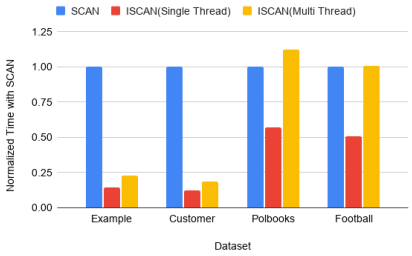

Figure 7: Comparision of run time of deletion by SCAN, Incremental SCAN and Incremental SCAN with multithreading algorithms on small datasets.

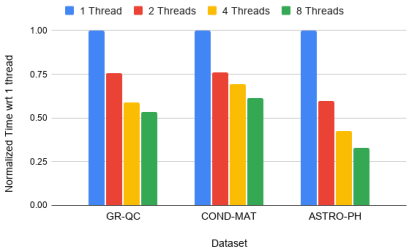

Figure 8: Multithreaded Performance of SCAN (on Big Datasets) with 0.7 epsilon.

these results. Clearly, the performance of incremental SCAN compared to normal improves with increase in graph size. Though the multi-threaded version performs better than SCAN, it still does not provide much improvement to single threaded incremental SCAN algorithm.

- **Deletion of random edges on large datasets:** We also ran the SCAN and incremental version on real world large datasets and deleted the last 500 edges. Here as well, we rerun SCAN after each edge deletion whereas update edge is called on incremental version. Results are summarized in Figure 4. For deletion as we can see incremental version outperforms SCAN by quite a margin.
- **Performance comparison of SCAN with multithreading:** We compared the run time of SCAN by changing the number of threads. Since the similarity calculations for SCAN are it's most computationaly expensive step hence parallelising this step provides significant gains. Based on the size of the dataset different number of threads can have different results on its runtime, these have been outlined in Figure 10 and Figure 8 .
- **Performance comparison by changing values of epsilon:** Finally we rerun certain experiments by varying the epsilon values. We have taken epsilon values to be 0.3, 0.5, 0.7 for the large datasets, since we don't know the ideal values for these graphs. We have analysed the running time of SCAN on the large graphs with varying epsilon in Figure 12, 13 and 14 .We ran the SCAN multi-threading vs without multi-threading after varying epsilon as seen in Figure 9, Figure 11 and Figure 8 . As we can observe in general on increasing the epsilon values the runtime of scan algorithm decreases.

From these graphs it is clear that our incremental SCAN implementation outperforms SCAN in running time analysis. This can be attributed to the lower time complexity of incremental SCAN in comparision with SCAN. SCAN has a time complexity of $O(m^{1.5})$ whereas Incremental SCAN has a worst case complexity of $O(m + nlogn)$. Incremental SCAN performs much better in real world graphs. These calculations are shown in section 4. Also one can see that multi-threading helps reduce the run time of SCAN algorithm. Though for incremental SCAN multi-threading is not so helpful in reducing run time. In fact the overhead of creating and forking of thread in addition to waiting time of threads caused multi-threaded version to have higher run time than single-threaded version. By this result we can conclude that out Incremental version of SCAN in fact performs significantly smaller number of update similarity values calculations.

6 LESSONS LEARNED FROM PROJECT

We have explored the SCAN algorithm for clustering graphs and its various variants implemented through the years through this project. This gave us a critical thinking task to analyze other authors' complex works and look for an unexplored niche. Also, implementing another author's work, however simple, presented several challenges, including but not limited to designing the code architecture, finding accurate datasets to replicate results, and trying to

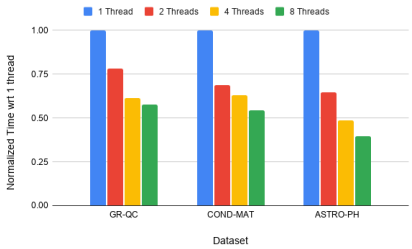

Figure 9: Multithreaded Performance of SCAN (on Big Datasets) with 0.3 epsilon.

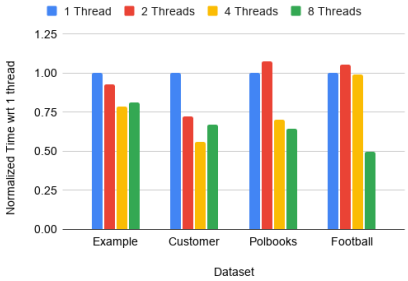

Figure 10: Multithreaded Performance of SCAN (on small Datasets) with their ideal epsilons.

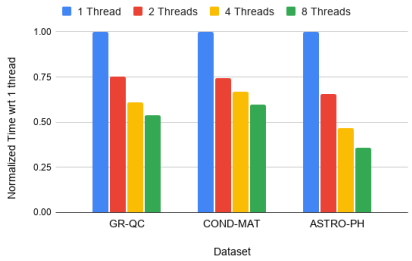

Figure 11: Multithreaded Performance of SCAN (on Big Datasets) with 0.5 epsilon.

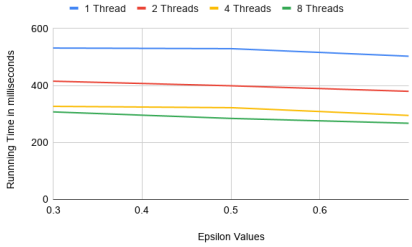

Figure 12: Running Time of SCAN on GR-QC with varying epsilon.

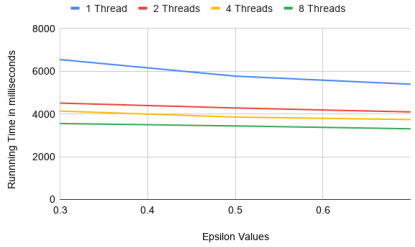

Figure 13: Running Time of SCAN on COND-MAT with varying epsilon.

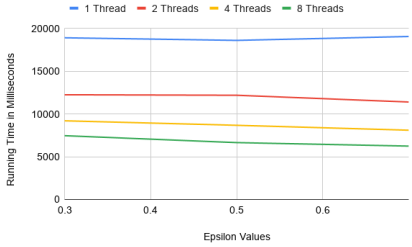

Figure 14: Running Time of SCAN on ASTRO-PH with varying epsilon.

tune parameters to get similar outputs for the datasets. Also, since the entire project was in C++, handling complex data structures using pointers often caused a hassle in debugging, and hence we learned new debugging techniques used for such large codebases.

After analyzing these algorithms, we designed an incremental variant of SCAN by keeping track of the bfs traversal trees and other essential edges and utilizing these to modify clusters on edge addition and deletion. Working out the edge cases in complex graphs problems was another significant stepping stone, and we learned to tackle it by individually coming up with such cases. We also constantly cross-questioned each other's proposals and found various flaws with our design, which we later fixed. A new area that we had previously not gotten the chance to explore was focusing on our algorithm's memory footprint and concurrency. Initially, our focus was on optimizing the time complexity, but we learned about the trade-off between run time and some other parameters like memory or accuracy.

We have shown a significant performance boost on both smaller manually annotated graphs and massive real-life graphs for both edge addition and deletion. We have also made the similarity calculation for the base SCAN algorithm parallel, which is the most computationally expensive step and shown its performance improvement over the baseline. We appreciate the opportunity to work on an algorithm of our choosing and have learned a lot more through this project than maybe a single theoretical course could have taught.

7 FUTURE WORK

We can formally implement Incremental SCAN as part of a open source library in different programming languages so that people can leverage the benefits of Incremental SCAN. Most of the modern-day data processing, owing to the scale of data, happens on distributed systems. We can modify our algorithm to work in scenarios where the graph in itself is stored over multiple systems. Another possible future avenue stems from the fact that the running time of Incremental SCAN allows us to generate real-time clustering results. Incremental SCAN can thus be used to create real-time dashboards for monitoring disease spread and other contact tracing applications.

Table 2: Project Summary

Link to base paper	SCAN
Link to code of base paper	Codebase
Link to datasets used	Example, customer, Football, Polbooks, GR-QC, COND-MAT, ASTRO-PH
Implemented Codebase	Codebase
Is code working	YES
Maximum Speed up	95% (average over big datasets)
Extra memory required	94% (worst case)
Accuracy	100%
How is accuracy measured	Comparing the clusters formed using SCAN and ISCAN
Interested in further improvements during summer	NO

REFERENCES

[1] 2006. College Football Standings. https://www.sports-reference.com/cfb/years/2006-standings.html. (2006).
[2] L. Chang, W. Li, X. Lin, L. Qin, and W. Zhang. 2016. pSCAN: Fast and exact structural graph clustering. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 253–264. https://doi.org/10.1109/ICDE.2016.7498245
[3] Yazhong Chen, Rong-Hua Li, Qiangqiang Dai, Zhenjun Li, Shaojie Qiao, and Rui Mao. 2017. Incremental Structural Clustering for Dynamic Networks. 123–134. https://doi.org/10.1007/978-3-319-68783-4_9

[4] Aaron Clauset, M Newman, and Cristopher Moore. 2005. Finding community structure in very large networks. *Physical review. E, Statistical, nonlinear, and soft matter physics* 70 (01 2005), 066111. https://doi.org/10.1103/PhysRevE.70.066111

[5] C. H. Q. Ding, Xiaofeng He, Hongyuan Zha, Ming Gu, and H. D. Simon. 2001. A min-max cut algorithm for graph partitioning and data clustering. In *Proceedings 2001 IEEE International Conference on Data Mining*. 107–114. https://doi.org/10.1109/ICDM.2001.989507

[6] Roger Guimerà and Luís A. Nunes Amaral. 2005. Functional cartography of complex metabolic networks. *Nature* 433, 7028 (01 Feb 2005), 895–900. https://doi.org/10.1038/nature03288

[7] J. Kleinberg J. Leskovec and C. Faloutsos. 2007. Astro Physics collaboration network. https://snap.stanford.edu/data/ca-AstroPh.html. (2007).

[8] J. Kleinberg J. Leskovec and C. Faloutsos. 2007. Condense Matter collaboration network. https://snap.stanford.edu/data/ca-CondMat.html. (2007).

[9] J. Kleinberg J. Leskovec and C. Faloutsos. 2007. General Relativity and Quantum Cosmology collaboration network. https://snap.stanford.edu/data/ca-GrQc.html. (2007).

[10] Jianbo Shi and J. Malik. 2000. Normalized cuts and image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22, 8 (2000), 888–905. https://doi.org/10.1109/34.868688

[11] Mark Newman. 2013. Mark Newman Net Datasets. http://www-personal.umich.edu/~mejn/netdata. (2013).

[12] M. E. J. Newman and M. Girvan. 2004. Finding and evaluating community structure in networks. *Physical Review E* 69, 2 (Feb 2004). https://doi.org/10.1103/physreve.69.026113

[13] Giulio Rossetti. 2019. CDLib Library. https://cdlib.readthedocs.io/en/latest/reference/cd_algorithms/algs/cdlib.algorithms.scan.html. (2019).

[14] Hiroaki Shiokawa, Yasuhiro Fujiwara, and Makoto Onizuka. 2015. SCAN++: Efficient Algorithm for Finding Clusters, Hubs and Outliers on Large-Scale Graphs. *Proc. VLDB Endow.* 8, 11 (July 2015), 1178–1189. https://doi.org/10.14778/2809974.2809980

[15] Dong Wen, Lu Qin, Ying Zhang, Lijun Chang, and Xuemin Lin. 2017. Efficient structural graph clustering: an index-based approach. *Proceedings of the VLDB Endowment* 11 (11 2017), 243–255. https://doi.org/10.14778/3157794.3157795

[16] Xiaowei Xu, Nurcan Yuruk, Zhidan Feng, and Thomas A. J. Schweiger. 2007. SCAN: A Structural Clustering Algorithm for Networks. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*. Association for Computing Machinery, New York, NY, USA, 824–833. https://doi.org/10.1145/1281192.1281280

[17] W. Zhao, G. Chen, and X. Xu. 2017. AnySCAN: An Efficient Anytime Framework with Active Learning for Large-Scale Network Clustering. In *2017 IEEE International Conference on Data Mining (ICDM)*. 665–674. https://doi.org/10.1109/ICDM.2017.76