
Graph Mining Using SQL

15-826 Final Project Report

Nijith Jacob
Carnegie Mellon University
njacob@andrew.cmu.edu

Sharif Doghmi
Carnegie Mellon University
sdoghmi@andrew.cmu.edu

Abstract

Is SQL powerful enough to do complex graph mining on real world datasets? In this paper, we show that this is in fact true by implementing various graph algorithms for computing the degree distributions, pagerank, connected components, radii of the vertices, eigendecomposition of the adjacency matrix, belief propagation and triangle counts. We also perform anomaly detection by extracting features from the egonet of the nodes. Finally, we apply these operations on 9 real world datasets and try to identify recurring and anomalous patterns in these graphs.

1 Introduction

Graph Mining is an active area of research in data mining that aims to discover patterns and anomalies in graphs by looking at their local and global properties. Since several real-world problems can be expressed as a graph problem, there is a lot of interest in mining useful information from them. From ranking pages in the world wide web, providing recommendations in social networks, analyzing biological pathways to detecting malicious activity in computer networks, applications of graph mining are diverse and interdisciplinary.

The most popular method for mining very large graphs is using a distributed MapReduce platform such as Hadoop. However, for moderately large graphs, SQL commands wrapped in a host programming language provides a much simpler and less expensive method of mining graphs stored in relational databases. A RDBMS like PostgreSQL provides query and storage optimization to facilitate mining large graphs with practical space and time complexity.

In this paper, we use SQL commands wrapped in Python programming language to interface with PostgreSQL RDBMS and implement various graph mining algorithms using matrix and vector operations on tabular graph data. The algorithms include finding degree distribution, pagerank, weakly connected components, node radii, eigendecomposition, belief propagation, and count of triangles. We have also implemented an additional task involving anomaly detection in graphs. We will use these implementations to perform broad-spectrum graph mining, where we apply our algorithms to many real world graph datasets in order to discover global patterns and detect anomalies.

In section 2, we summarize related work. In section 3, we describe the algorithm and method behind the various mining tasks. In Section 4, we describe our graph mining experiments and provide the analysis of the results. Finally, in section 5 we present our conclusions based on the results we obtained.

2 Survey

In this section, we list the surveyed papers.

2.1 Papers Surveyed by Nijith Jacob

2.1.1 OddBall: Spotting Anomalies in Weighted Graphs [1]

Main Idea

In this paper, anomalous nodes are detected by characterizing some features of its neighbourhood. Here the neighbourhood is chosen as the egonet or the induced subgraph of the 1-step neighbours of a node.

The two main questions this paper addresses are

1. How to choose features for the neighbourhood? and
2. What pattern or law characterize a normal neighbourhood?

The paper narrows down the set of "interesting" features to

1. Number of nodes/degree of the node (N)
2. Number of edges in the egonet (E)
3. Total weight of edges (W)
4. Principal eigenvalue of the adjacency matrix of the egonet (λ)

The interesting observation that is employed in detecting outliers is that several real-world graphs obey a power law between certain pairs of these features. This reduces the problem of anomaly detection to finding the nodes that deviate from the power law for a particular graph.

Based on the power law violated, different types of anomaly can be detected such as near-Cliques, stars, dominant pairs, heavy vicinity etc.

Relevance to project

I'm planning to explore the ideas in this paper for the extra task.

Limitations

The method here is useful only when the features follow a power law. Though the law has been observed in several real world graphs, it will be an interesting problem to find the characteristics of graphs where such laws are violated.

2.1.2 EigenSpokes: Surprising Patterns and Scalable Community Chipping in Large Graphs [2]

Main Idea

This paper presents a surprising and recurring pattern (termed by the authors as EigenSpokes) in large sparse social graphs like mobile phone calls, internet, patent citations etc. The pattern can be used to identify community like structures in large sparse graphs. In particular, two community structures are identified namely cliques and near-perfect bipartite cores.

Their method is different from traditional methods like spectral clustering and graph partitioning methods in that the focus is not on partitioning the entire graph. Rather, the focus is on identifying close knit communities that could be loosely connected to an otherwise random graph core.

The primary method they employ is the singular value decomposition of the adjacency matrix to identify community like structures in the graph. The intuition behind this is that SVD decomposition gives the principal components and nodes belonging to a community typically have similar adjacent nodes and hence similar components along the

singular vectors. The surprising observation here is that such nodes in sparse and large graphs tend to have large components along one of the singular vector.

The algorithm that the paper explores greedily add connected nodes having large components along a singular vector until adding more nodes decrease the quality score of the community detected.

Relevance to project

In undirected graphs, the adjacency matrix is symmetric and therefore the singular vectors are the same as eigenvectors. Since one of the tasks for the project is eigenvalue decomposition of the adjacency matrix, the pattern discovered in this paper can be easily applied as part of the project to discover community like structures.

Limitations

The idea expressed in this paper is limited to large sparse graphs. So this approach may not work when the graph is dense.

2.1.3 GBASE: an efficient analysis platform for large graphs [3]

Main Idea

This paper proposes an efficient analysis platform for very large scale graphs. In particular, the paper proposes efficient storage mechanisms, common core algorithms and an optimized query execution build on top of HADOOP.

For efficient graph storage, a novel method called 'compressed block encoding' is used. The community like structures common in real world graphs are exploited to form a more efficient storage mechanism by first partitioning the graph into homogeneous blocks. Each block represented by the adjacency matrix of the induced subgraph of a pair of partitioned nodes is compressed using standard algorithms. Finally, a grid placement of the blocks into different files is performed to optimize both in-neighbours and out-neighbours queries.

An important observation mentioned in the paper is the generalization of common graph operations as matrix-vector multiplication where the matrix can be the adjacency matrix or the incidence matrix. Since matrix-vector multiplication can be easily mapped into SQL joins, it can be easily and efficiently implemented using modern database systems.

The query execution is built on top of HADOOP and MapReduce framework where the matrix-vector multiplication is carried out over the relevant blocks for a particular graph operation.

Relevance to project

The idea of expressing common graph problems as matrix-vector multiplication or equivalently as a SQL join is central to the project. In addition this suggests that the various tasks to be implemented in the project has reusable and common primitives/operations that can be used for writing modular code.

Limitations

This approach may not work for graph operations that cannot be expressed as some form of matrix-vector multiplication.

2.2 Papers Surveyed by Sharif Doghmi

2.2.1 PEGASUS: Mining Peta-Scale Graphs [4]

Main Idea

This paper describes PeGaSus, an open source Peta Graph Mining library implemented on top of the Hadoop platform that performs many graph mining operations. PeGaSus uses mainly a single primitive called Generalized Iterative Matrix-Vector Multiplication (GIM-V), which is a generalized form of matrix-vector multiplication. GIM-V is composed of three main operations that can be customized to perform various graph mining operations, including

PageRank, node proximity measurement, diameter estimation, and finding connected components.

A new algorithm, HCC, was proposed for finding connected components in large graphs.

GIM-V can be performed naively or with performance optimizations. The following table lists different algorithms for performing GIM-V. Performance analysis was performed for the algorithms and they are listed ordered by performance from lowest to highest

Algorithm	Takes advantage of
GIM-V Base	Nothing (naïve method)
GIM-V CL	Clustered edges
GIM-V BL	Block multiplication
GIM-V BL-CL	Block multiplication and Clustered edges

In addition, there are two more algorithms for GIM-V that can be used to decrease the number of iterations needed when using HCC to find connected components. GIM-V DI uses repeated diagonal block multiplications within each iteration. GIM-V NR rennumbers a center node with the minimum ID.

Finally, GIM-V was performed on real world graphs to examine connected components, PageRank, and graph diameter. In agreement with previous research findings, power-law relations and stabilization after the gelling point were observed. Anomalies were detected.

Relevance to project

GIM-V can be implemented using SQL statements with user-defined functions in combination with another programming language to enable iterations. Typical GIM-V graph mining operations can be performed with the performance and ease of use advantages of SQL. Analysis methods of real world graphs in this paper can be used as a starting point for our analyses. Findings in the paper can be compared to our findings.

Limitations

Since we have no control over the internal workings of SQL commands, optimized GIM-V algorithms cannot be taken advantage of. Also, the graph mining operations performed in this paper were done on Hadoop and using parallel computations on many machines. Since we will be using SQL on a single system, this limits the performance of our operations for large data sets and the size and number of the graphs we can analyze.

2.2.2 Mining Large Graphs: Algorithms, Inference, and Discoveries [5]

Main Idea

The paper describes a new efficient parallel algorithm called Hadoop Line Graph Fixed Point (HA-LFP) to implement belief propagation on large graphs and make inferences about node states. In particular, it finds the most probable distribution of node states. It is implemented on Hadoop which uses the programming model of MapReduce, providing performance and scalability advantages, and is particularly valuable for graphs that do not fit in memory.

The algorithm first requires that the original graph be converted to a line graph, whose nodes correspond to edges in the original graph. The algorithm runs for as many iterations needed to reach convergence. In each iteration, each node passes messages to its adjacent nodes about its beliefs, updating a global message vector. Convergence is reached after the message vector stops changing with iterations. After convergence, inferences can be made about the states of the nodes in the graph.

In the paper, the algorithm was applied to analyze real-world graphs in different ways such as to identify 'good web pages and bad web pages on the internet, to identify roles of people in social networks and to identify suspicious and anomalous nodes in real world graphs using several different methods.

Relevance to project

The paper explains the use of an algorithm that uses a primitive very similar to the GIM-V primitive discussed in the previous paper, for which SQL code was already written. It is a stepping stone to learning how to apply the binary

Fast" belief propagation algorithm needed for the assignment to analyze various graphs using belief propagation. Our findings can be compared to and contrasted with findings of the paper.

Limitations

The algorithm is computationally expensive. It was designed with parallel computation in mind and might have performance limitations when run on a single machine with SQL code. It only finds probability distributions of states and cannot draw absolute conclusions about nodes.

2.2.3 Unifying Guilt-by-Association Approaches: Theorems and Fast Algorithms [6]

Main Idea

Guilt-by-Association (GbA) methods can be applied to real-world graphs in order to predict classifications of unknown nodes based on classifications of known nodes, also known as prior beliefs about those nodes. This process is called belief propagation. Several GbA methods exist: Random Walk with Restarts, Semi-Supervised Learning, and Belief Propagation (BP). The paper describes a new algorithm called Fast Belief Propagation (FaBP) that is similar to BP, but faster, with equal or higher accuracy, and is guaranteed to converge.

All the four previous methods can be expressed as a linear system expressed as a matrix multiplication of the form: $M \times b = p$, where M is an $n \times n$ matrix (n is number of nodes) unique to that method, b is a vector of the final propagated beliefs of the nodes, and p is a vector of the initial beliefs. All four methods are related and some of them will produce identical results using certain values of constants within M . This linear system can be solved by multiplication of the p vector by the the inverse of M . This is computationally intractable for very large matrices, so the paper suggests using the power method for FaBP and describes its application.

The paper also describes how to select c' and a , two constants in M in a way to guarantee convergence of the power method for FaBP. They are calculated from a quantity called the about half homophily factor (h_h). This factor is taken from the maximum of two values calculated based on the 1-norm and the Frobenius norm of a matrix derived from M .

Experiments were conducted using FaBP on real world graphs. In the experiments, it was found that FaBP made the same predictions as BP. It was shown that the propagated beliefs converge when the homophily factor chosen is within a certain bound. Otherwise, they diverge. FaBP was implemented on HADOOP and shown to be linearly scalable on the number of edges. It was also about twice as fast as BP.

Relevance to project

FaBP will be implemented using SQL statements to apply the power method to real world datasets to propagate beliefs and visualize the outcomes. SQL on Postgres is much more spatially efficient for large graphs than actual matrix multiplication using a package like MATLAB. The latter is practically impossible to use for very large graphs.

Limitations

Availability of datasets with prior beliefs on nodes is limited. Thus, we will be flexible in our experiments using the datasets that are available. Plausible scenarios in real-world datasets will be examined.

3 Method

This section describes the various algorithms and SQL code used for the completion of the project.

3.1 Conventions used

For convenience and clarity, the following conventions are used when describing the algorithms and SQL codes.

1. Table signatures

Table signatures specify the general column format of the frequently used tables.

- **Adjacency Table:** *dgraph (src_id integer, dst_id integer, weight real)*
- **Matrix:** *matrix (row_id integer, col_id integer, value real)*
- **Vector:** *vector (id integer, value real)*

2. Notational conventions:

The following are the notational conventions used in describing the algorithms. The matrix and vector operations mentioned are easily implemented over tables with the table signatures described above using the SQL codes in section 3.2.

- $A \times B$: matrix multiplication of table A and table B .
- $u \cdot v$: dot product between vector table u and vector table v
- $\|\cdot\|_2$: L2 norm of vector
- $A \leftarrow B$: Insert all rows in B to A after truncating A
- $Q(i, j)$: value at i^{th} row and j^{th} column.
 $Q(:, j)$: j^{th} column vector.
- $EIG(A)$: Eigen decomposition of A
- n : number of nodes in the graph
- $dgraph(A)$: Adjacency table for directed graphs
- $graph(A_u)$: Adjacency table for undirected graphs

3.2 Generic SQL codes

This section contain implementations of common SQL codes that accomplish operations that are used in the various algorithms.

1. Euclidean(L2) norm of vector

```
SELECT sqrt(sum((value)^2)) FROM vector
```

2. Vector dot product ($u \cdot v$)

```
SELECT sum(vector1.value * vector2.value)
FROM vector1, vector2
WHERE vector1.id = vector2.id
```

3. Matrix multiplication ($A \times B$)

```
SELECT A.row_id, B.col_id, sum(A.value * B.value)
FROM mat1 "A", mat2 "B"
WHERE A.col_id = B.row_id
GROUP BY A.row_id, B.col_id
```

4. Adjacency table \times vector

```
SELECT A.src_id, sum(A.weight * V.value)
FROM graph "A", vector "V"
WHERE A.dst_id = V.id
GROUP BY A.src_id
```

5. Obtaining graph from *dgraph*

```
INSERT INTO graph(src_id, dst_id, weight)
SELECT src_id, dst_id, weight FROM dgraph
UNION ALL
SELECT dst_id "src_id", src_id "dst_id", weight FROM dgraph
```

3.3 Graph mining algorithms

This section describes the graph mining algorithms that are used for the various project and extra tasks. The SQL code is provided where appropriate. The algorithms for connected components, graph radius, graph eigenvalue, belief propagation, triangle count and anomaly detection work on the undirected version of the graph.

3.3.1 Degree Distribution

The various degrees of a node are computed as the number of rows in the adjacency table with $src_id = node_id$ (out degree), $dst_id = node_id$ (in degree) and $src_id = node_id$ or $dst_id = node_id$ (degree). For an undirected graph all three degrees are equal. The distribution for each of the degrees refer to the frequency distribution of the degree across all nodes in the graph.

Algorithm: Degree Distribution

- **In-Degree Distribution**

```
SELECT indegree, count(*) "count" FROM
  (SELECT count(*) "indegree" FROM dgraph GROUP BY dst_id)
GROUP BY indegree
```

- **Out-Degree Distribution**

```
SELECT outdegree, count(*) "count" FROM
  (SELECT count(*) "outdegree" FROM dgraph GROUP BY src_id)
GROUP BY outdegree
```

- **Degree Distribution**

```
SELECT degree, count(*) "count" FROM
  (SELECT count(*) "degree" FROM dgraph GROUP BY dst_id
   UNION ALL
   SELECT count(*) "degree" FROM dgraph GROUP BY src_id)
GROUP BY degree
```

3.3.2 PageRank

PageRank is the famous algorithm that was originally employed on Google web search engine to rank webpages. The pagerank score is determined both by the number of links to a page as well as the ranking of the pages linked from. The algorithm finds the fixed point probabilistic distribution of ranks across all the pages (given by the dominant eigenvector) using the power method.

To account for disconnected pages, links are added from a page to every other page including itself. The resulting graph can be best described in the context of a random surfer who follows links across pages with probability proportional to the weight of the link and also at times probabilistically stop and restart from a random node.

Algorithm: PageRank

Input:

1. damping factor (c) = 0.85 : The damping factor controls the probability of restart for a random surfer to restart the random walk from another node.
2. Stopping threshold (ϵ), maximum number of iterations (m)

Output:

1. $pagerank(node_id, page_rank), p$: Probabilistic ranking of every node in the graph.

Auxiliary tables:

1. $offset_table(node_id, page_rank), p_o$: This is the offset that is added during each pageRank iteration.
2. $pagerank_next(node_id, page_rank), p_{new}$: This vector stores the next iteration of pageRank
3. $norm_table(src_id, dst_id, weight), A_{norm}$: Row normalized version of adjacency table.

Algorithm:

1. Initialize:

- (a) Compute row normalized adjacency table, *norm_table*

```
INSERT INTO norm_table
  SELECT src_id, dst_id, weight/weight_sm "weight"
  FROM dgraph "TAB1",
  (SELECT src_id "node_id", sum(weight) "weight_sm"
  FROM dgraph
  GROUP BY src_id) "TAB2"
  WHERE "TAB1".src_id = "TAB2".node_id
```

- (b) Initialize *offset_table* to $(1 - c)/n$

- (c) Initialize *pagerank* to $1/n$

- (d) *step* = 1

2. Compute next Pagerank:

The next iteration of PageRank values is computed as $p_{new} = cA_{norm}^T p + \alpha p_o$ where α is the sum of pageranks in *p*.

```
INSERT INTO pagerank_next
  SELECT node_id, SUM(pagerank)
  FROM (
    SELECT dst_id "node_id", SUM(0.85*weight*page_rank) "page_rank"
    FROM norm_table, pagerank
    WHERE src_id = node_id GROUP BY dst_id
  UNION ALL
  SELECT node_id, page_rank * val "page_rank"
  FROM offset_table, (SELECT SUM(pagerank) "val" FROM pagerank)
  )
  GROUP BY node_id
```

3. Iterate until Convergence:

If $\|p_{new} - p\|_2 > \epsilon$ and *step* $\leq m$, set $p \leftarrow p_{new}$, *step* = *step* + 1, goto step 2. Otherwise set $p \leftarrow p_{new}$ and break iteration.

3.3.3 Weakly Connected Components

The weakly connected components of a graph are the maximal undirected subgraphs such that within a component all nodes are reachable from any other node. The algorithm described below to find such components work by keeping track of the component (identified by a component id) to which each node belongs. First we initialize such that each node belongs to a component of its own. We then iterate, updating the component to which each node belongs. At the k^{th} iteration, we set a node's component id as the minimum of the ids of all nodes that are at most k hops away (the choice of minimum here is completely arbitrary. We can use any function that determines a unique value from a set of values). The algorithm will eventually converge when k equals the maximum radii of the graph at which the neighbourhood around each node would have expanded to include all the reachable nodes.

Algorithm: Weakly Connected Components

Output:

1. $conn_comp(node_id, component_id)$, c : The component id is a unique id determined as the minimum of the ids of the nodes within a neighbourhood around a node.

Auxiliary tables:

1. $conn_comp_new(node_id, component_id)$, c_{new} : The component ids at the next iteration.

Algorithm:

1. Initialize:

Each node's component id is initialized to the node's id.

2. Iterate:

Each node's component id is set as the
 $\min \{ \text{node's component id, component ids of its neighbours} \}$

```
INSERT INTO conn_comp_new
  SELECT node_id, MIN(component_id) "component_id"
  FROM (
    SELECT src_id "node_id", MIN(component_id) "component_id"
      FROM graph, conn_comp
     WHERE dst_id = node_id
    GROUP BY src_id
    UNION
    SELECT * FROM conn_comp
  )
  GROUP BY node_id
```

3. Iterate until convergence:

If $\|c - c_{new}\|_2 > 0$, $c \leftarrow c_{new}$, goto step 2. Otherwise break iteration.

3.3.4 Radius of Every Node

Martin-Flajolet method for calculating node radii uses repeated bit-wise OR's of probabilistic bitstrings of neighboring nodes to approximate the size of reachable neighbourhood of the nodes. The bitstring of a node obtained at the k^{th} iteration is an approximation of the number of nodes k hops away. The intuition behind this method of approximating size of a set is that the bit strings are encoded in such a way that the i^{th} bit is set with probability $\frac{1}{2^{i+1}}$. The index of the leftmost 1, r in the bitstring during convergence is therefore a good indicator of the number of reachable nodes and is given by $\frac{1}{0.77351} 2^{r+1}$ (after normalizing). The effective radius (the radius at which at least 90% of reachable nodes lie) of the nodes are computed from this bitstring as the iteration number at which the number of neighbours obtained from the bitstring is within a threshold (90%) of the maximum number of neighbours approximated at convergence.

Algorithm: Radius of Every Node

Input:

1. maximum number of steps (m)

Output:

1. $node_radius(node_id, radius)$: The effective radius of all nodes.

Auxiliary tables:

1. $hop_table_{\{i\}}(node_id, bit_string)$: Bit string hash for every node after i iterations. h_i will be used to refer to the vector of bitstrings in this table.
2. $max_neighbourhood_table(id, value)$: The neighbourhood value at the maximum hop.

Algorithm:

1. Initialize:

- (a) Bit strings in hop_table_0 is initialized as the output from function given by $\frac{1}{2}(hash(node_id) \text{ xor } (hash(node_id) - 1) + 1)$. This function zeros out all but the rightmost 1 in the hash value. The hash function $hash()$ used for this implementation is given by $hash(node_id) = (node_id \bmod n) + 1$ where n is the number of nodes. This initialization can be easily performed in SQL by selecting this functional expression from a table containing the node ids.

2. Iterate for $i = 1$ to m steps:

(a) Compute next hop bitstrings:

The next hop bitstrings are computed as *bitwise-or* of a node's bitstring with that of its neighbours. This is accomplished by the following SQL.

```
INSERT INTO hop_table_{i}
```

```

SELECT node_id, bit_or(bit_string) FROM
  (SELECT src_id "node_id", bit_or(bit_string) "bit_string"
   FROM graph, hop_table_{i-1}
   WHERE dst_id = node_id GROUP BY src_id
  UNION ALL
  SELECT * FROM hop_table_{i-1})
GROUP BY node_id

```

(b) **Check for convergence:**

If $\|h_i - h_{i-1}\|_2 = 0$, set $max_hop = i$ and break iteration.

3. **Compute neighbourhood value at max hop:**

The neighbourhood value is computed as $\frac{1}{0.77351} 2^R$ where R is the index of leftmost 0 given by $\lfloor \log_2(bit_string) + 1 \rfloor$

```

INSERT INTO max_neighbourhood_table
  SELECT node_id "id", 2^(floor(log(2,bit_string)+1))/0.77351 "value"
  FROM node_table_{max_hop}

```

4. **Iterate for $i = 0$ to max_hop steps:**

(a) **Get nodes with effective radius i :**

A node's effective radius is taken as i if the value of the neighbourhood function computed on $hop_table.i$ is ≥ 0.9 times the value at max hop. This is accomplished by the following SQL

```

INSERT INTO node_radius
  SELECT node_id, {i} "radius"
  FROM hop_table_{i}, max_neighbourhood_table
  WHERE node_id = id
  AND 2^(floor(log(2,bit_string)+1))/0.77351 >= 0.9*value

```

(b) To avoid inserting as radius all values from i to max_hop for converged nodes, the newly inserted nodes are deleted from the $max_neighbourhood_table$.

```

DELETE FROM max_neighbourhood_table
  WHERE id IN (SELECT node_id FROM node_radius)

```

3.3.5 Graph Eigenvalues

Eigenvalues of the adjacency matrix are fundamental in analysis of the spectral properties of graphs. They can also be used to find approximations to important graph measures such as triangle counts. Since the eigenvalues of non-symmetric graphs can be complex, only undirected graphs are dealt with here. The algorithm discussed here is the Lanczos algorithm with selective orthogonalization.

Note: Since the algorithms discussed in this section heavily use matrix operations, the SQL statement for each step is not provided for clarity. Please refer to the section on generic SQL codes for implementations of these matrix operations in SQL.

1. **Lanczos Algorithm with Selective Orthogonalization**

The basic Lanczos algorithm is an iterative algorithm to find the top k eigenvalues of a matrix. The algorithm works by orthogonalizing the set of intermediate vectors obtained during power iteration to form a basis for the matrix (A_u). At each iteration, a tridiagonal matrix T is constructed that is similar to A_u such that $T = B^T A_u B$ where B is the set of basis vectors constructed so far. We can then approximate the eigenvalues and eigenvectors of A , from the eigendecomposition of T . This can be performed easily by the QR algorithm (discussed next).

The problem with the basis Lanczos algorithm is that the basis vectors lose their orthogonality due to finite precision. Since full reorthogonalization with all the previous basis vectors can be expensive, Lanczos selective orthogonalization is performed where we look for eigenvectors that has converged and reorthogonalize with respect to them.

Algorithm: Lanczos Selective Orthogonalization

Input:

- (a) m : the number of steps to iterate
 - m_{qr} : the number of steps to iterate for the QR algorithm
- (b) ϵ_1 : threshold used for selective orthogonalization.
 - ϵ_2 : threshold used to end iteration
 - ϵ_{qr} : threshold used for the QR algorithm

Output:

- (a) $eigval_table(\mathbf{A}')$: The eigenvalue vector table.
- (b) $eigvec_table(\mathbf{U}')$: The eigenvector table. The eigenvectors are the column vectors of this table.

Auxiliary tables:

- (a) $basis_vect_0(v_0)$: The $i - 1^{th}$ basis vector.
- (b) $basis_vect_1(v_1)$: The i^{th} basis vector.
- (c) $basis_vect_new(v_{new})$: The $i + 1^{th}$ basis vector.
- (d) $tri_diag_table(\mathbf{T})$: Symmetric tridiagonal matrix similar to A_u
- (e) $basis_table(\mathbf{B})$: Table with column vectors as the basis vectors upto iteration i

Algorithm:

- (a) **Initialize:**
 - v_0 is initialized to 0
 - v_1 is initialized randomly and then normalized
 - $\beta_0 = 0$
- (b) **Iterate for $i = 1$ to m steps:**
 - i. **Get new basis vector:** $v_{new} = A_u \times v_1$
 - ii. $\alpha_1 = v_{new} \cdot v_1$
 - iii. **Orthogonalize:** Orthogonalize the new basis vector against previous two basis vector using the Gram-Schmidt method.
$$v_{new} \leftarrow v_{new} - \alpha_1 v_1 - \beta_0 v_0$$

```
SELECT "VECT_NEW".id,  
      ("VECT_NEW".value-(b0*"VECT0".value)-(a1*"VECT1".value)) "value"  
FROM basis_vect_new "VECT_NEW",  
      basis_vect_0 "VECT0",  
      basis_vect_1 "VECT1"  
WHERE "VECT_NEW".id = "VECT0".id AND "VECT0".id = "VECT1".id
```
 - iv. **Build the tridiagonal table:**
 $\mathbf{T}(i, i) = \alpha_1, \mathbf{T}(i - 1, i) = \beta_0, \mathbf{T}(i, i - 1) = \beta_0$
 - v. **Build the basis table:**
 $\mathbf{B}(:, i) \leftarrow v_1$
 - vi. Get the eigen decomposition of \mathbf{T} using QR algorithm.
 $\mathbf{U}\mathbf{\Lambda}\mathbf{U}^T \leftarrow EIG(\mathbf{T})$
 - vii. **Iterate for $j = 1$ to i :**
 - A. Check whether the j^{th} eigenvector has converged.
Is $abs(\mathbf{U}(i, j)) < \epsilon_1$? Then goto next step. Otherwise continue iteration on j .
 - B. Get the j^{th} eigenvector
 $r = \mathbf{B} \times \mathbf{U}(:, j)$
 - C. Do selective orthogonalization with this eigenvector.
$$v_{new} \leftarrow v_{new} - (r \cdot v_{new}) r$$
 - viii. $\beta_1 = \|v_{new}\|_2$
 - ix. Break iteration if the new basis vector is in the span of previous basis vectors. i.e. if $\beta_1 < \epsilon_2$

- x. Normalize v_{new} .
- xi. **Prepare for next iteration:**
 $v_0 \leftarrow v_1, v_1 \leftarrow v_{new}, \beta_0 = \beta_1$
- (c) $\mathbf{U}\mathbf{\Lambda}\mathbf{U}^T \leftarrow EIG(\mathbf{T})$
- (d) **Get the eigenvalues:** $\mathbf{\Lambda}' \leftarrow diag(\mathbf{\Lambda})$
- (e) **Get the eigenvectors:** $\mathbf{U}' \leftarrow \mathbf{B} \times \mathbf{U}$

2. QR Algorithm

QR algorithm is an iterative algorithm to calculate the eigendecomposition of a matrix. At each iteration we compute $A_{k+1} = R_k Q_k = Q_k^T A_k Q_k$ where $A_k = Q_k R_k$ is the QR decomposition of the matrix. The intuition behind this algorithm is that A_{k+1} is similar to A_k and hence have the same eigenvalues. The eigenvectors after m iterations can be computed as $Q_1 Q_2 \dots Q_m$.

The general QR decomposition algorithm is expensive. Since we are concerned only with tridiagonal matrices, QR decomposition can be adapted to run in linear time (discussed next).

Algorithm: QR Algorithm

Input:

- (a) $tri_diag_table(\mathbf{T})$: Symmetric tridiagonal table
- (b) m_{qr} : the maximum number of steps to iterate
- (c) ϵ_{qr} : stopping threshold

Output:

- (a) $eigval_table(\mathbf{\Lambda})$: The eigenvalue table.
- (b) $eigvec_table(\mathbf{U})$: The eigenvector table.

Algorithm:

- (a) **Initialize:** $\mathbf{V} \leftarrow \mathbf{T}, \mathbf{U} \leftarrow \mathbf{I}$, where \mathbf{I} is the identity matrix
- (b) **Iterate for $i = 1$ to m_{qr} steps:**
 - i. **QR decomposition:**
 Compute QR decomposition which satisfies $\mathbf{V} = \mathbf{Q} \times \mathbf{R}$
 - ii. $\mathbf{V} \leftarrow \mathbf{R} \times \mathbf{Q}$
 - iii. $\mathbf{U} \leftarrow \mathbf{U} \times \mathbf{Q}$
 - iv. **Check threshold:** Is $max(abs(\mathbf{V}_{non-diagonal})) < \epsilon_{qr}$? Then break iteration. Otherwise continue.
Rationale: The maximum non-diagonal element of \mathbf{V} is a good indicator of the convergence of diagonal eigenvalues.
- (c) $\mathbf{\Lambda} \leftarrow \mathbf{V}$

3. QR Decomposition

QR decomposition is the decomposition of a matrix T such that $T = QR$ where Q is an orthogonal matrix and R is an upper triangular matrix. The algorithm discussed here is an adaptation to the special case when T is a tridiagonal matrix. The intuition behind this algorithm is that T being nearly upper triangular, we can iterately set the below diagonal elements to 0 by using the appropriate *Givens rotation matrix*.

Algorithm: QR Decomposition

Input:

- (a) $tri_diag_table(\mathbf{T})$: Symmetric tridiagonal table

Output:

- (a) $Q_table(\mathbf{Q})$
- (b) $R_table(\mathbf{R})$

Algorithm:

- (a) **Initialize:** $\mathbf{R} \leftarrow \mathbf{T}, \mathbf{Q} \leftarrow \mathbf{I}$, where \mathbf{I} is the identity matrix

- (b) **Iterate for $i = 1$ to $n - 1$ steps** (where n is the size of \mathbf{T}):
- i. **Compute Givens rotation matrix:**
 Let $\alpha = \mathbf{R}(i, i)$ and $\beta = \mathbf{R}(i + 1, i)$
 Let $r = \sqrt{\alpha^2 + \beta^2}$, $c = \alpha/r$, $s = -\beta/r$
 $\mathbf{G} \leftarrow \mathbf{I}$, $\mathbf{G}(i, i) = c$, $\mathbf{G}(i + 1, i + 1) = c$, $\mathbf{G}(i, i + 1) = -s$, $\mathbf{G}(i + 1, i) = s$
 - ii. $\mathbf{Q} \leftarrow \mathbf{Q} \times \mathbf{G}^T$
 - iii. $\mathbf{R} \leftarrow \mathbf{G} \times \mathbf{R}$
-

3.3.6 Fast Belief Propagation

Belief propagation algorithms are used to find the steady state beliefs of the graph given a set of prior beliefs on the nodes of the graph. Here we discuss Fast Belief Propagation (FaBP) algorithm that propagate beliefs across a graph faster than other belief propagation algorithms and with high accuracy.

The algorithm starts with a set of prior beliefs (ϕ_h) on the nodes such that good nodes have positive belief value, bad nodes have negative belief value and neutral/unknown nodes have their belief value set at 0. Prior beliefs are then propagated to the unknown nodes to spot good and bad nodes. FaBP converges very quickly after only a few iterations. FaBP uses a carefully chosen constant called the about-half homophily factor (h_h) to determine the strength of the similarity in beliefs (speed of diffusion) between adjacent nodes. The homophily factor affects the speed of the convergence. An incorrectly chosen homophily factor can lead to beliefs diverging and the algorithm never ending.

More formally, FaBP solves a linear system of equations given by $[I + aD - c'A]b_h = \phi_h$ using the power method. Here a and c' are chosen based on the homophily factor derived from the graph. D is a diagonal matrix of node degrees, A is the adjacency matrix and b_h are the about-half unknown beliefs on the graph. The solution involves the inversion of a matrix of the form $I - W$ where $W = c'A - aD$. The inverse is obtained by performing power iteration on W with the next set of belief values computed by $b_{h,next} = Wb_h + \phi_h$.

Algorithm: Fast Belief Propagation

Input:

1. *prior_belief* (*node_id*, *belief*), ϕ_h : The prior beliefs on the graph
2. Stopping threshold (ϵ), maximum number of iterations (m)

Output:

1. *belief* (*node_id*, *belief*), b_h : The final set of beliefs on the graph.

Auxiliary tables:

1. *node_degrees* (*node_id*, *degree*), d : The degrees of each node.
2. *belief_next* (*node_id*, *belief*), $b_{h,next}$: The next set of beliefs on the graph

Algorithm:

1. Initialize:

- (a) Get the node degrees.

```
INSERT INTO node_degrees
SELECT node_id, count(*) "degree"
FROM graph
GROUP BY node_id
```

- (b) Initialize belief table. $b_h \leftarrow \phi_h$

- (c) Get the best homophily factor (h_h) and from it the constants c' and a .

$$h_h = \max\left(\frac{1}{2 + 2\max(d_{ii})}, \sqrt{\frac{-c_1 + \sqrt{c_1^2 + 4c_2}}{8c_2}}\right) \text{ where } c_1 = 2 + \sum d_{ii}, c_2 = \sum d_{ii}^2 - 1$$

The constants are computed as $a = \frac{4h_h^2}{1 - 4h_h^2}$ and $c' = \frac{2h_h}{1 - 4h_h^2}$

In SQL, these can be easily computed from the *node_degrees* table.

(d) *step* = 1

2. Compute next beliefs:

The following SQL compute the next set of beliefs.

```
INSERT INTO belief_next
SELECT node_id, SUM(belief) "belief"
FROM
  (SELECT src_id "node_id", {c'}*SUM(belief) "belief"
   FROM graph, belief
   WHERE dst_id = node_id
   GROUP BY src_id
  UNION ALL
   SELECT nodeid "node_id", {-a}*degree*belief "belief"
   FROM node_degrees "d", belief "b"
   WHERE "d".node_id = "b".node_id
  UNION ALL
   SELECT node_id, belief FROM prior_belief)
GROUP BY node_id
```

3. Iterate until Convergence:

If $\|b_{h,next} - b_h\|_2 > \epsilon$ and $step \leq m$, set $b_h \leftarrow b_{h,next}$, $step = step + 1$, goto step 2. Otherwise set $b_h \leftarrow b_{h,next}$ and break iteration.

3.3.7 Count of triangles

The count of triangles is an important measure in a graph. It can be used to spot anomalous graphs and abnormal nodes. The naive method of computing the count of triangles would be to use $\frac{trace(A_u^3)}{6}$. The division by 6 is done since we count the same triangle 3! times. But this becomes intractable fast for large graphs. Therefore, we will use an approximate algorithm that use eigenvalues to compute the triangle count. The intuition is that trace of a matrix is equal to the sum of its eigenvalues and that eigenvalues of A_u^k are λ_i^k where λ_i are the eigenvalues of A_u . For large graphs this sum can be approximated by the top k eigenvalues obtained using eigendecomposition algorithm discussed earlier.

Algorithm: Triangle Count

Output:

1. Approximate number of triangles

Algorithm:

1. Eigen decomposition:

Use eigen decomposition algorithm to compute the top k eigenvalues of the graph

$$\Lambda \leftarrow EIG(graph)$$

- 2. Count:** The count of triangles can be approximated as $\frac{\sum \lambda_i^3}{6}$

```
SELECT sum(value^3)/6 FROM eigval_table
```

3.3.8 Anomaly Detection in Weighted Graphs (Extra)

To detect anomalies in a graph, we use the ideas mentioned in the paper "OddBall: Spotting Anomalies in Weighted Graphs" [1] (surveyed in section 2.1.1). The outliers are detected by characterizing the normal behavior in the graph by a power law and using that to quantify the measure by which they deviate. As suggested in the paper, we extract three features from the egonet of each node namely, the number of nodes (*N*), number of edges (*E*) and the total

weight of edges(W). The characterizing power laws between E and N and between W and E are found by linear fitting. An outlier score is then given to each node proportional to their distance from the best fit line. Finally, nodes with high score are reported as anomalies.

The power law between E and N detect anomalies like star configuration ($E = N - 1$) on one extrema to a perfect clique ($E = N(N - 1)$) on the other extrema. The power law between W and E find anomalies where the weight in the egonet is really large compared to the number of edges (Heavy vicinity). In this paper, we will be working with undirected graphs.

Algorithm: Anomaly Detection

For the proposed method, we need to extract three features from the egonet of each node. The number of nodes N is equal to the degree of the node + 1. This can be easily extracted from the adjacency table. The number of edges (E) and the total weight of edges (W) for each egonet are extracted in 2 steps. In the first step, for each node, we add the edges that are incident on its neighbours. This can be accomplished by joining the adjacency table to itself twice. Since we are working with undirected graphs, this step would add the same edge twice which need to be corrected appropriately. In the second step, we add all the edges that are incident on the node. The set of edges thus obtained from these two steps represent the egonet of the nodes and the number of edges and the sum of its weight can be extracted from this set.

This can be accomplished by using the following SQL which returns the node's id, the count of edges (edge_cnt) and sum of edge weights (wgt_sum)

```
SELECT node_id, sum(edge_cnt) "edge_cnt", sum(wgt_sum) "wgt_sum"
FROM
  (SELECT "T2".dst_id "node_id", count(*)/2 "edge_cnt",
        sum("T2".weight)/2 "wgt_sum"
   FROM graph "T1", graph "T2", graph "T3"
   WHERE "T1".src_id = "T2".src_id
        AND "T1".dst_id = "T3".dst_id
        AND "T2".dst_id="T3".src_id
   GROUP BY "T2".dst_id)
UNION ALL
SELECT src_id "node_id", count(*) "edge_cnt", sum(weight) "wgt_sum"
FROM graph)
GROUP BY node_id
```

3.4 Implementation Summary

The following are the specific implementation details for our project

- Database: Postgres
- Programming/Scripting Environment: Python
- Libraries:
 1. psycopg2: python library for interfacing with postgres

All the data processing is performed in SQL. The scripting environment is used for interfacing and for basic loop and conditional constructs.

The output for each of the mining algorithm is stored on the database in an appropriate table.

For more details, please refer to the documentation provided with the software packaging.

4 Experimental Analysis

This section is divided into 2 subsections. In the first, we elucidate on the various unit tests that were done to verify the correctness of the implementation. In the second, we give a thorough analysis of the results obtained on various real-world datasets.

4.1 Correctness Verification

In addition to the unit tests done at the time of coding, the correctness of the algorithms implemented as part of this project were verified by comparing the results with their implementations in Matlab. The implementation code in Matlab is provided along with the software packaging. For most of the algorithms, the dataset *Advogato*¹ was used for verification. The dataset is a weighted and directed graph with 6, 551 nodes and 51, 332 edges.

1. Degree Distribution

Figure 1 compares the in-degree, out-degree and degree distributions obtained using SQL and Matlab. The plots are slightly shifted for visual clarity.

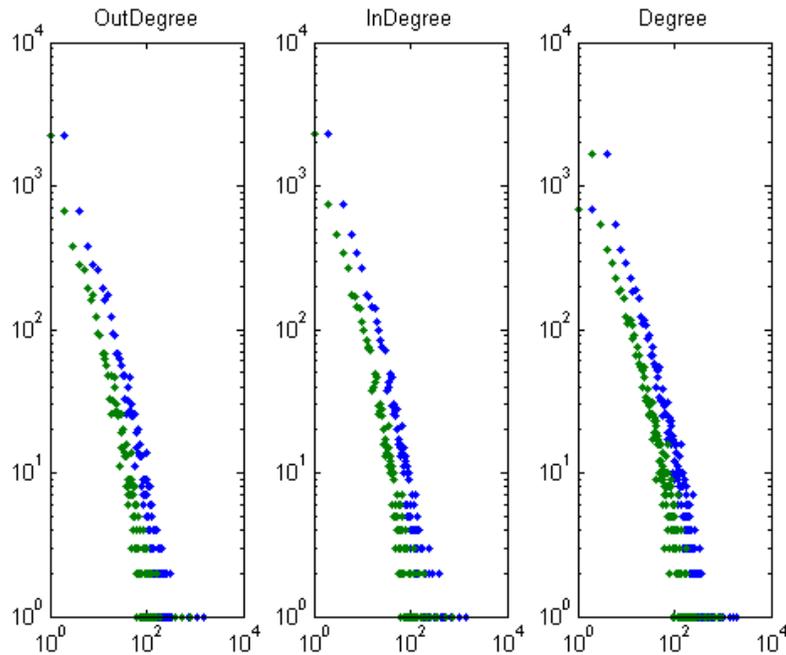


Figure 1: Out-degree, In-degree and degree distributions for implementation in Matlab(Blue) and in SQL(Green)

2. PageRank

PageRank was implemented in Matlab by using *eigs()* to obtain the dominant eigenvector. The parameters used for the implementation in SQL were *damping_factor* = 0.85, *max_iterations* = 10, *stop_threshold* = 0.01. Figure 2 plots the pageranks obtained from the 2 implementations. Note: The scaling factor on the axes are different since the two pagerank vectors are normalized differently (pagerank obtained has L1 norm as 1 while the one obtained as the eigenvector has L2 norm as 1).

3. Weakly Connected Components

¹<http://konect.uni-koblenz.de/networks/advogato>

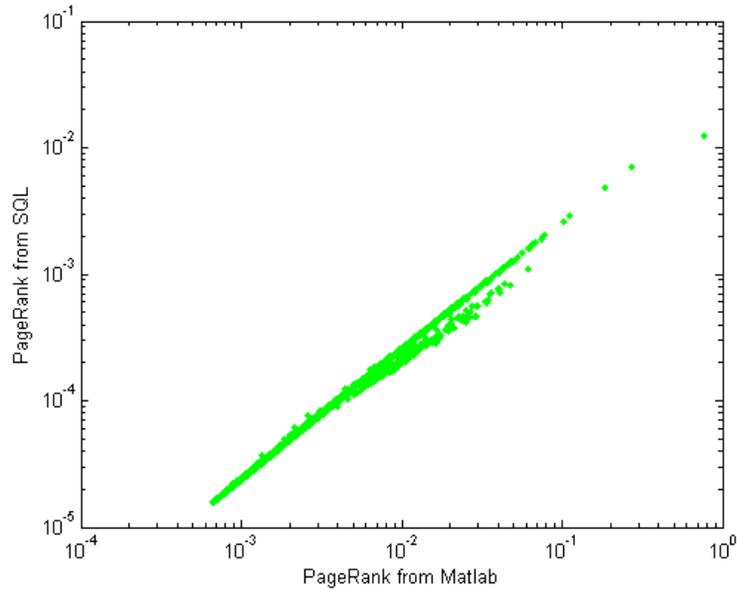


Figure 2: Scatter plot of pageranks obtained from SQL and Matlab

Same algorithms were used in both the SQL and Matlab implementations. Figure 3 is the frequency plot of component sizes.

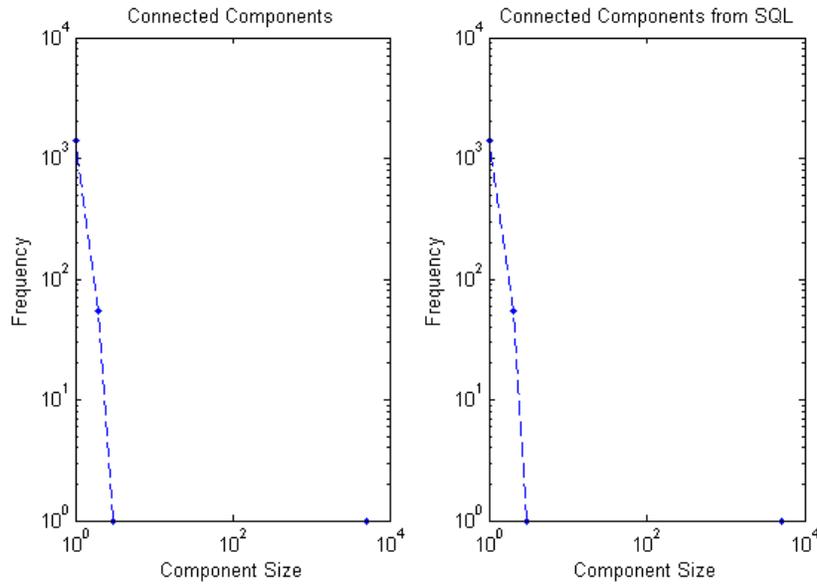


Figure 3: Frequency plot for component sizes (log-log) for implementation in Matlab(Left) and in SQL(Right)

4. Radius of every node

The Matlab implementation returns the exact radius for each node. The exact radius was computed by keeping track of the k-hop neighbours. The SQL implementation is approximate and it returns the effective radius computed as the radii at which 90% of the reachable neighbours are seen. Figure 4 is the histogram distribu-

tion of the radius. The figures are a little different since the SQL implementation is approximate, calculates the effective radius and is dependent on the the choice of hash function. But, it is clear that both the results follow similar distribtution of radius.

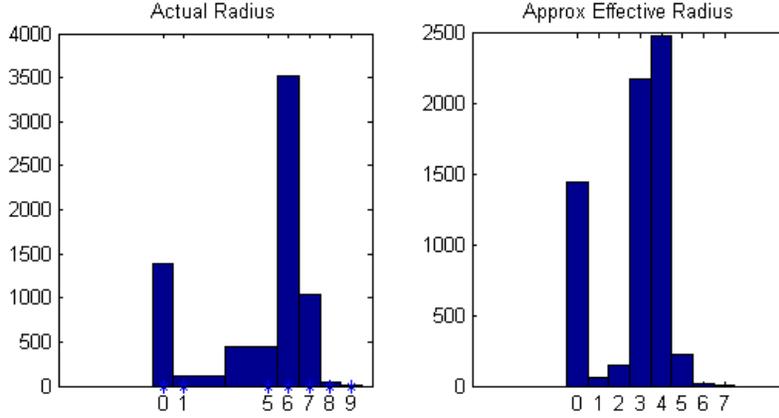


Figure 4: Histogram of radius for implementation in Matlab(Left) and in SQL(Right)

5. Eigenvalues

We verify the Lanczos-SO algorithm and QR deomposition algorithm using the following matrix \mathbf{T} both as an adjacency matrix (for eigendecomposition) and as a tridiagonal matrix (for QR decomposition).

$$\mathbf{T} = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}$$

(a) QR Decomposition

The Q and R obtained were exactly the same, given by

$$\mathbf{Q} = \begin{bmatrix} -0.7071 & 0 & -0.4082 & -0.2887 & 0 & 0.5000 \\ -0.7071 & 0 & 0.4082 & 0.2887 & 0 & -0.5000 \\ 0 & -1.0000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.8165 & -0.2887 & 0 & 0.5000 \\ 0 & 0 & 0 & -0.8660 & 0 & -0.5000 \\ 0 & 0 & 0 & 0 & -1.0000 & 0 \end{bmatrix}$$

$$\mathbf{R} = \begin{bmatrix} -1.4142 & -1.4142 & -0.7071 & 0 & 0 & 0 \\ 0 & -1.0000 & -1.0000 & -1.0000 & 0 & 0 \\ 0 & 0 & 1.2247 & 0.8165 & 0.8165 & 0 \\ 0 & 0 & 0 & -1.1547 & -1.1547 & -0.8660 \\ 0 & 0 & 0 & 0 & -1.0000 & -1.0000 \\ 0 & 0 & 0 & 0 & 0 & -0.5000 \end{bmatrix}$$

(b) Eigendecomposition

- Eigenvalues(Λ) and eigenvectors(U) obtained for T from Matlab using $eig()$

$$\mathbf{U} = \begin{bmatrix} -0.2319 & -0.4179 & 0.5211 & 0.5211 & -0.4179 & 0.2319 \\ 0.4179 & 0.5211 & -0.2319 & 0.2319 & -0.5211 & 0.4179 \\ -0.5211 & -0.2319 & -0.4179 & -0.4179 & -0.2319 & 0.5211 \\ 0.5211 & -0.2319 & 0.4179 & -0.4179 & 0.2319 & 0.5211 \\ -0.4179 & 0.5211 & 0.2319 & 0.2319 & 0.5211 & 0.4179 \\ 0.2319 & -0.4179 & -0.5211 & 0.5211 & 0.4179 & 0.2319 \end{bmatrix}$$

$$\Lambda = [-0.8019 \quad -0.2470 \quad 0.5550 \quad 1.4450 \quad 2.2470 \quad 2.8019]$$

- Eigenvalues(Λ) and eigenvectors(U) obtained for T using Lanczos-SO algorithm(rounded)

$$U = \begin{bmatrix} 0.2319 & -0.4174 & -0.5215 & 0.2319 & 0.5211 & 0.4179 \\ 0.4179 & -0.5209 & -0.2324 & -0.4179 & -0.2319 & -0.5211 \\ 0.5211 & -0.2323 & 0.4177 & 0.5211 & -0.4179 & 0.2319 \\ 0.5211 & 0.2315 & 0.4181 & -0.5211 & 0.4179 & 0.2319 \\ 0.4179 & 0.5214 & -0.2314 & 0.4179 & 0.2319 & -0.5211 \\ 0.2319 & 0.4184 & -0.5207 & -0.2319 & -0.5211 & 0.4179 \end{bmatrix}$$

$$\Lambda = [2.8019 \quad 2.2470 \quad 1.4450 \quad -0.8019 \quad 0.5550 \quad -0.2470]$$

6. Fast Belief Propagation

Prior beliefs were assigned randomly to nodes in the *Advogato* dataset. 10% were positive nodes with assigned prior beliefs = +0.01. 30% were negative nodes with prior beliefs = -0.01. The rest had prior beliefs = 0. The beliefs were propagated using both SQL and MATLAB and shown in Figure 5. To the left, is the initial scatter plot of beliefs. All beliefs are either 0, -0.1, or +0.1. To the right is the scatter plot of final beliefs for SQL (y-axis) vs. MATLAB (x-axis). Notice how both predicted the same beliefs. All points lie on the $y=x$ line and there are no differently predicted nodes in the upper left or lower right quadrants. Also notice that more nodes were shifted to the negative side than to the positive because there were more nodes with prior negative beliefs than positive.

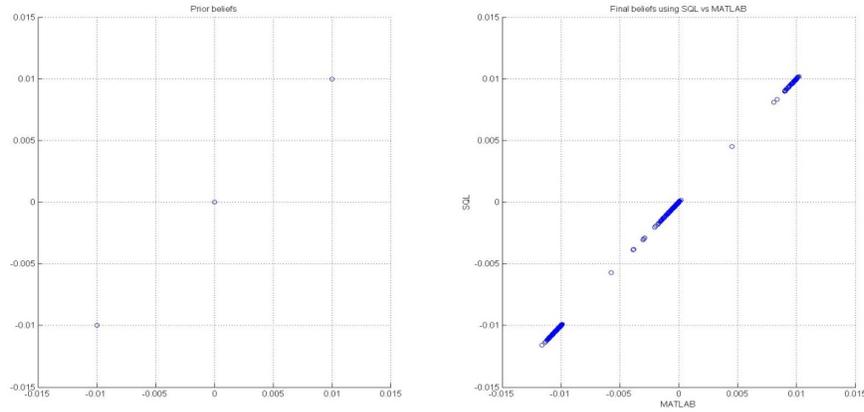


Figure 5: Prior belief scatter plot (Left) and the propagated belief scatter plot (Right)

7. Count of triangles

For this, a smaller dataset, *Florida ecosystem*² was used. The correctness was verified by implementing a naive algorithm in SQL which computed the exact number of triangles as $\frac{\text{trace}(A^3)}{6}$.

- Number of triangles (naive) = 8715.0
- Number of triangles (approx from eigenvalues) = 8583.17

8. Anomaly Detection in weighted graphs (Extra)

Similar algorithms were implemented in Matlab and SQL and the plots obtained are shown in Figure 6

²<http://konect.uni-koblenz.de/networks/foodweb-baydry>

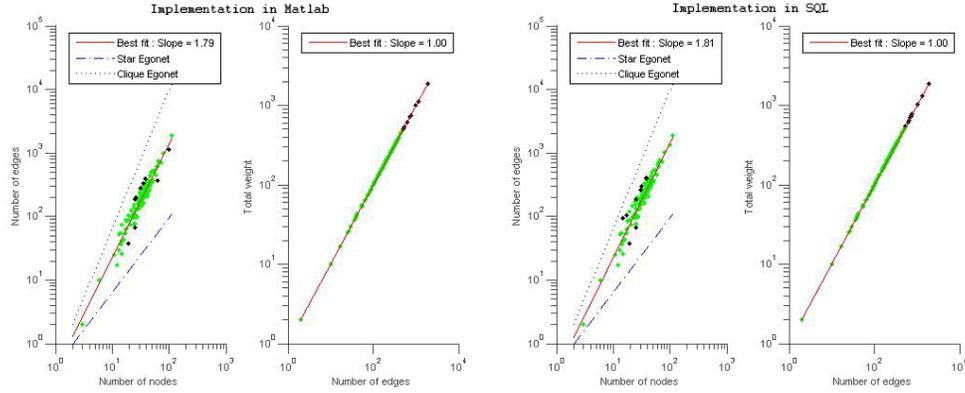


Figure 6: Anomaly Detection graphs for implementations in Matlab(Left) and in SQL(Right)

4.2 Graph Mining Results on real world datasets

Graph mining algorithms discussed earlier were applied on a number of real world datasets and in this section we elaborate and infer some conclusions on the patterns that emerged. We present our analysis on distinctive patterns for each graph algorithm. The graphs were selected from different categories to identify similarities within and across the various categories. We discuss the results on 5 graph categories: *Social*, *Co-occurrence*, *Reference*, *Ratings* and *Physical*. The datasets used for the graph experiments are shown below.

Dataset	Category	Directed	Weighted	Size	Edge Count	Description
Twitter ³	Social	Y	N	465K	835K	twitter who follows whom
Youtube ⁴	Social	N	N	3.2M	12.2M	youtube friendship connections
Amazon ⁵	Co-occurrence	Y	N	403K	3.4M	bought X also bought Y
Flickr ⁶	Co-occurrence	N	N	106K	2.3M	images sharing metadata
Google ⁷	Reference	Y	N	876K	5.1M	web hyperlinks
Patent citation ⁸	Reference	Y	N	3.8M	16.5M	US patent citation
Amazon ⁹	Ratings	Y	Y	3.4M	5.8M	bipartite user product ratings
Stack Overflow ¹⁰	Ratings	Y	N	642K	1.3M	bipartite favourite posts of users
Skitter ¹¹	Physical	N	N	1.7M	11M	autonomous systems on web

4.2.1 Results Overview

Here we present a summary of the results obtained on the various datasets. For *In degree*, *Out degree*, *PageRank* and *Connected Components*, the table indicate whether the result obeyed power law. For *Eigendecomposition*, the result indicate whether the dominant eigenvector pairs formed EigenSpokes[2] pattern. \boxtimes is used for a positive result, \boxminus for a negative result and \boxplus for intermediate/not sure results.

³http://konect.uni-koblenz.de/networks/munmun_twitter_social

⁴<http://konect.uni-koblenz.de/networks/youtube-u-growth>

⁵<http://konect.uni-koblenz.de/networks/amazon0601>

⁶<http://konect.uni-koblenz.de/networks/flickrEdges>

⁷<http://konect.uni-koblenz.de/networks/web-Google>

⁸<http://snap.stanford.edu/data/cit-Patents.html>

⁹<http://konect.uni-koblenz.de/networks/amazon-ratings>

¹⁰<http://konect.uni-koblenz.de/networks/stackexchange-stackoverflow>

¹¹<http://konect.uni-koblenz.de/networks/as-skitter>

Algorithm	Dataset									
	Twitter	Youtube	Amazon	Flickr	Google	Patents	Amazon Ratings	SO	Skitter	
In Degree	☑	☑	☑	☐	☑	☑	☑	☑	☑	☑
Out Degree	☒	☑	☒	☐	☑	☑	☑	☑	☑	☑
PageRank	☐	☑	☑	☑	☑	☑	☑	☑	☑	☑
ConnComp	☑	☑	☑	☑	☑	☑	☑	☑	☑	☑
Eigenvector	☑	☒	☒	☑	☑	☑	☑	☐	☑	☒

4.2.2 Degree distributions

In-degree distribution (See Figure 7) and Out-degree distribution (See Figure 8) of most of the graphs follow the power law. But for some of the graphs, we see a radical departure in the pattern for out-degree distribution in particular. We discuss these below.

1. **Twitter Social:** We see that a large fraction of twitter population have zero out-degree (following no one) implying that they are probably inactive. On the other hand, it is interesting to note that everyone is followed by atleast one other user. From the in-degree distribution, we see that the maximum number of followers for any user is 199. From the out-degree distribution we see that the count of active users (followers of someone) for a given out-degree (number of users followed) remains somewhat constant (< 10) until we see a sharp rise at around 400. These represent over active users and we see that their population size is markedly distinct from the rest. We can also spot two outlier users having more than 600 followees.
2. **Amazon Co-occurrence:** The in-degree distribution follows a power law showing that really good products are very rare and that there is a large number of bad/new products in the market. The out-degree distribution provides some insight into the shopping pattern of users. We see that the maximum out-degree is 10 implying that the users buy from a set of 10 related products (probably because they are the top 10 products in that category).
3. **Flickr Co-occurrence:** The degree distribution is discontinuous at degree 5 and degree 100. The degree value 100 is very interesting. This probably means that flickr users tend to upload images (having similar tags and location data) in batches of size 100.

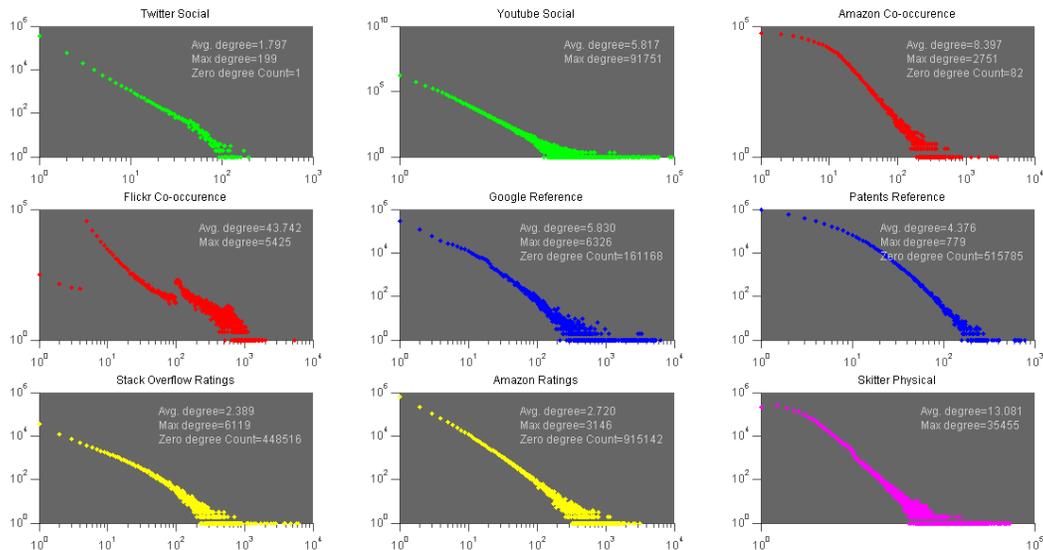


Figure 7: In-degree distribution of the datasets

4.2.3 PageRank

We analyzed the pageRank score and the number of nodes with that score (See Figure 9). We observe a power relation between them in almost all of the graphs. In the **Twitter Social** graph we see a sharper increase in the count for low

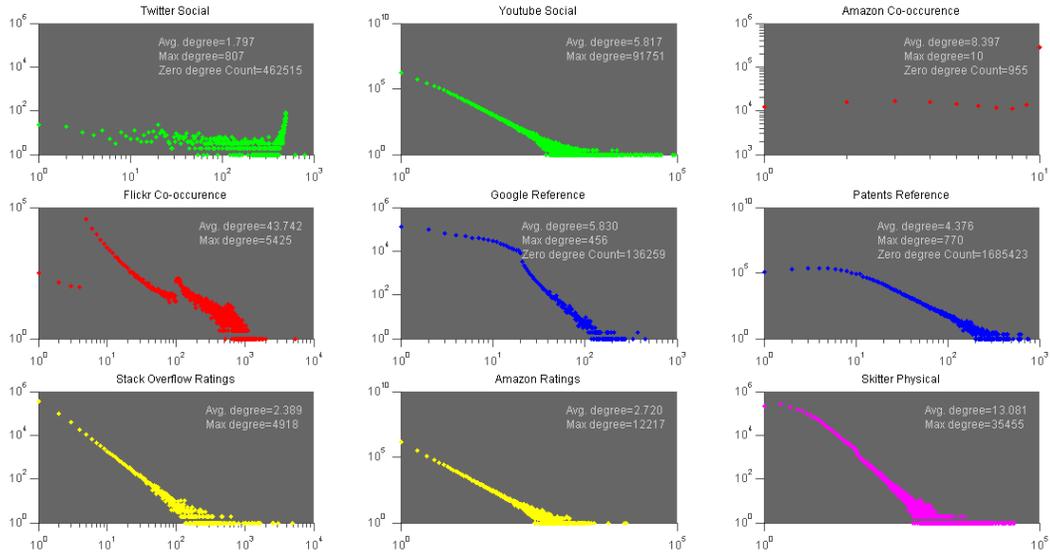


Figure 8: Out-degree distribution of the datasets

pageRank scores. This can be supported by the observation from Figure 8 that a large fraction of twitter users have very low degrees.

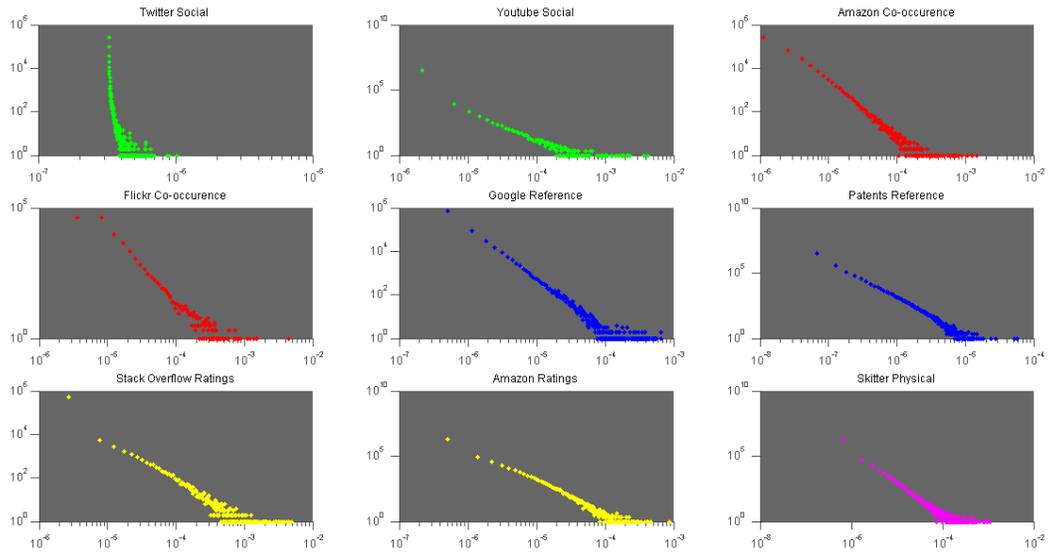


Figure 9: PageRank distribution of the datasets

4.2.4 Weakly connected components

We analyze the component size and the number of components with that size (See Figure 10). We can observe a power relation for components of small sizes. In all the graphs, we see that either the graph is fully connected or has a giant component relative to the sizes of the rest of the components.

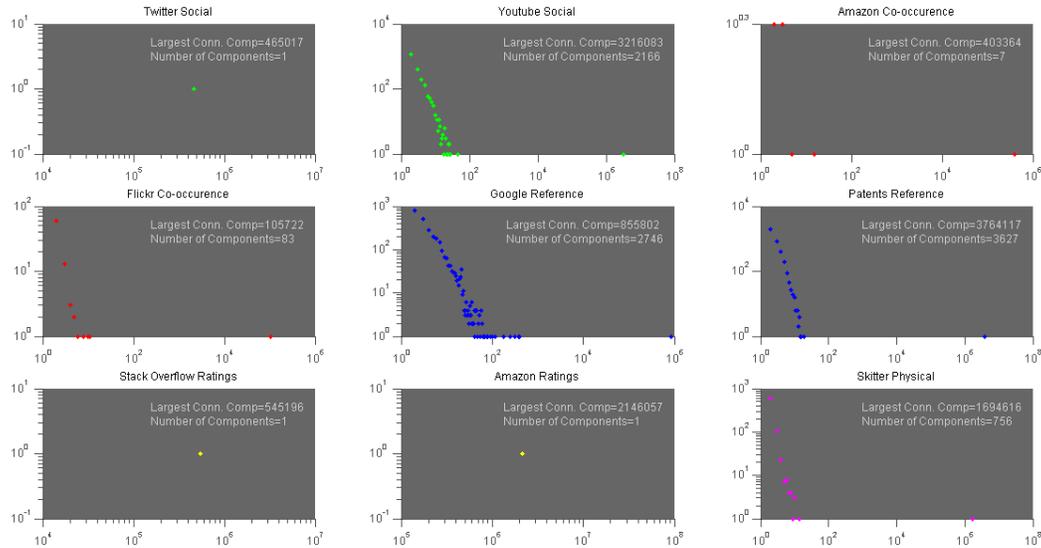


Figure 10: Connected Component distribution of the datasets

4.2.5 Graph Radius

We analyzed the radius distribution of the nodes in the graph (See Figure 11). We observed that the average node radius lie in the range of 4.578 and 7.442 showing that real world datasets have small diameter and are well connected. We also observe a bimodal/unimodal distribution. This is supported by the observation that was made on the connected components in a graph. The first mode at low radius values comes from nodes belonging to the smaller components whose distribution follow a power law. The second mode near the average radius comes from nodes belonging to the giant component.

In Figure 11, we observe that datasets **Twitter Social**, **Amazon Co-occurrence**, **Stack Overflow Ratings** and **Amazon Ratings** are near unimodal distributions. These datasets are fully connected or have very few small components as shown in Figure 10. The rest of the datasets follow a bimodal distribution of radius.

4.2.6 Eigendecomposition of Graphs

We analyzed the eigenvectors of the adjacency matrix of the graph (See Figure 12) using the method described in the paper on EigenSpokes[2]. This paper is surveyed in section 2.1.2. We plotted the least correlated pair amongst the top 4 eigenvectors returned from the Lanczos-SO algorithm. Nodes having strong values along one of the spokes may belong to near cliques and near bipartite cores.

We see that except for **Youtube Social**, **Amazon Co-occurrence** and **Skitter Physical**, the rest of the graphs have spoke patterns along the axes. For some of the graphs, some plausible explanations can be given. **Flickr Co-occurrence** has a very strong pattern and can be explained by the observation that the edge relationship in this graph is somewhat transitive resulting in cliques. **Youtube Social** and **Skitter Physical** shows very striking similarities and the significant departure from the pattern observed in the rest of the graphs suggest a different graph structure.

4.2.7 Fast Belief Propagation

The fast belief propagation algorithm was run on the *DLBP*¹² dataset. The edges in the graph connect authors who have published at least one paper together. Prior beliefs were assigned to authors based on their participation in conferences. Authors in certain conferences were assigned to a positive class, while those in other conferences were assigned to a negative class. The rest of the authors had unknown priors. These prior beliefs were then propagated to other authors,

¹²<http://snap.stanford.edu/data/com-DBLP.html>

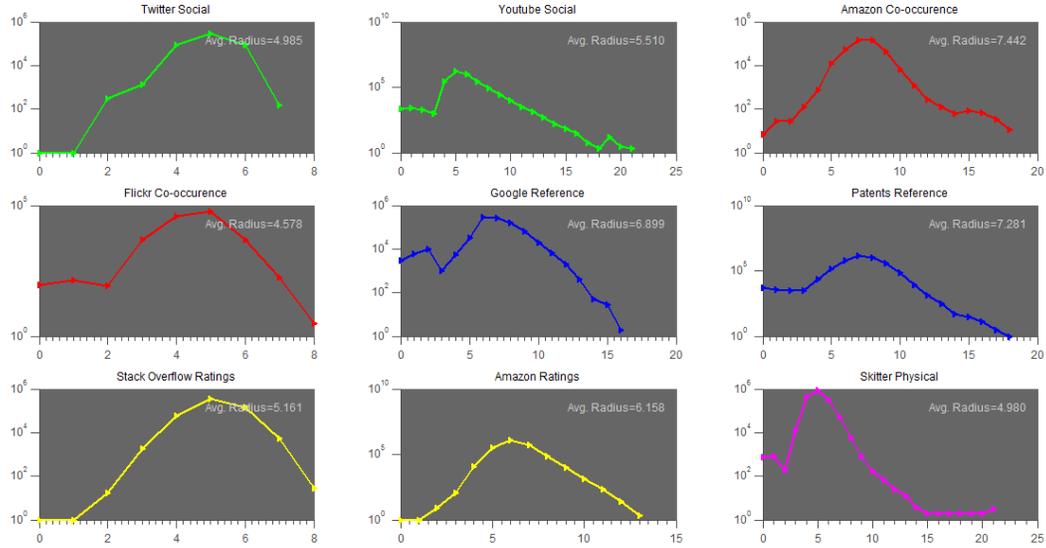


Figure 11: Radius distribution of the datasets

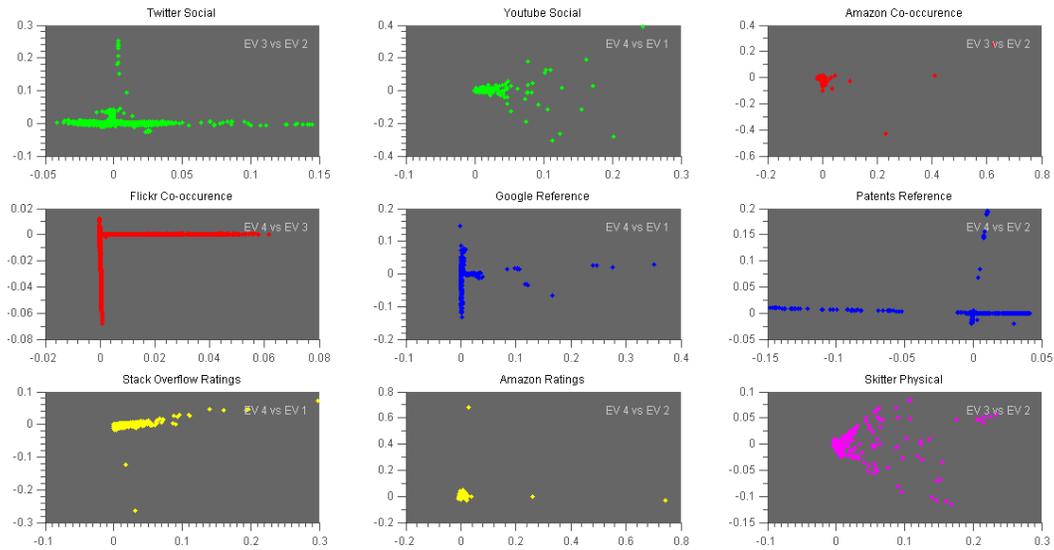


Figure 12: Plot of least correlated pair among the 4 top eigenvectors

with neighboring authors affecting each other most. Figure 13 shows five different distributions of final propagated beliefs under five different prior belief distributions corresponding to five different scenarios.

1. **Scenario 1:** Authors participating in the largest conference (the one with the most authors) got prior beliefs of $+0.001$. These authors represented about 2.5% of total authors. On the other hand, those participating in the third largest conference got prior beliefs of -0.001 . The two conferences were unrelated and had little overlap in authors. The rest of the authors were assigned zero (unknown) priors.

Results: There were more positive authors after propagation because there were more to start with. For the same reason, more unknown authors shifted towards the positive side than towards the negative side. Very few authors crossed over completely to either side because of the relatively small percentage of labelled authors, reducing their

ability to influence other authors, and due to the small maximum homophily factor allowed to ensure convergence in this setting.

2. **Scenario 2:** As in scenario 1, authors participating in the largest conference got prior beliefs of $+0.001$. This time authors participating in the second largest conference and not in the first were assigned prior beliefs of -0.001 . Unlike the first scenario, there was significant overlap in authors participating in both conferences.
Results: The final distribution was more equal on both sides than in scenario 1. Even though the negative conference had less authors, more unknown authors shifted to the negative side. Upon examination of the graph, this can be attributed to the second conference being better connected to the rest of the graph than the first.
3. **Scenario 3:** There were only positive beliefs assigned in this scenario. Authors participating in the biggest conference were given prior beliefs of $+0.001$. There were no negative prior beliefs.
Results: As expected, unknown authors shifted only to the positive side.
4. **Scenario 4:** Authors participating in the top three conferences were assigned positive prior beliefs of $+0.001$. There were no negative prior beliefs.
Results: Beliefs after propagation for unknown authors were noticeably more positive than in scenario 3. Beliefs of positive authors increased even more than in scenario 3 because there were more of them, and they boosted the beliefs of each other more, with no counteraction by negative authors in this case.
5. **Scenario 5:** Same as scenario 1 but with positive authors given the much more positive prior beliefs of $+0.1$, and negative authors given the much more negative prior beliefs of -0.1 .
Results: The distribution of beliefs after propagation is identical in shape to that of figure 1, except for being on a different x-axis scale. This shows that the shape of the final distribution depends less on the actual magnitude of prior beliefs and more on the distribution of prior beliefs and the shape of the graph.

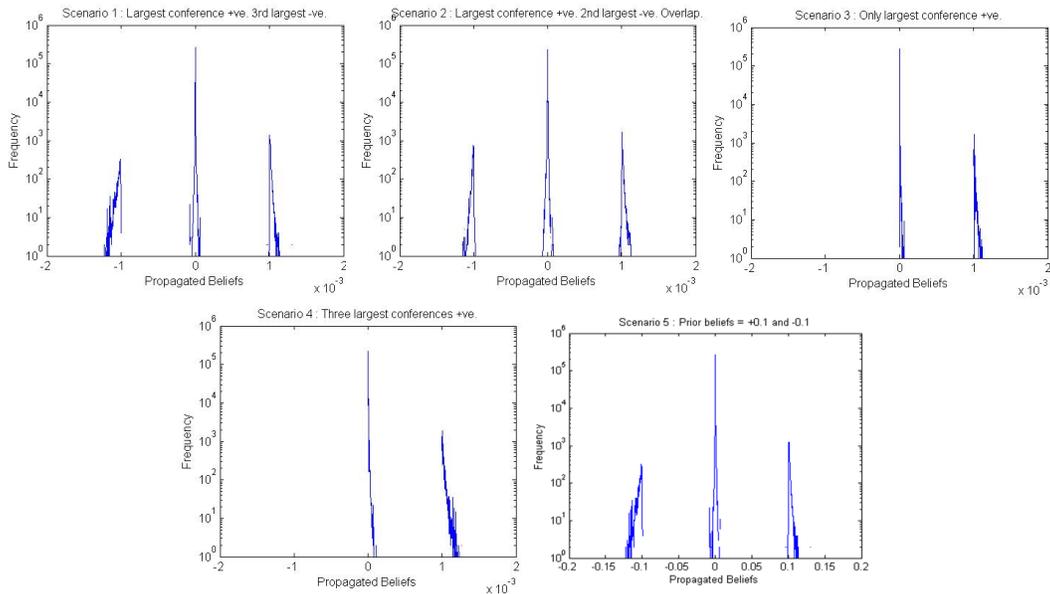


Figure 13: Propagated beliefs under 5 different prior belief assignments.

4.2.8 Count of Triangles

The number of triangles approximated from the eigenvalues are shown in Figure 14. The huge number of triangles for **Flickr Co-occurrence** indicate a strongly connected graph and also supports the observation we made in Figure 12. The high triangle count in **Youtube Social** is expected since friends of friends are usually friends themselves.

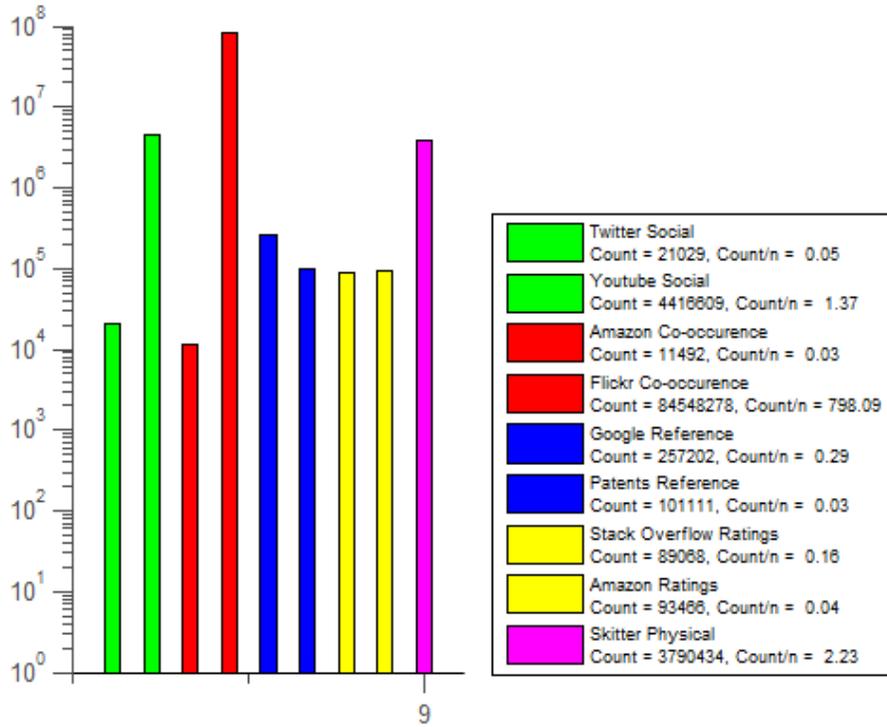


Figure 14: Count of triangles for each of the graphs. The count and normalized count (count/num of nodes) are given.

4.2.9 Anomaly Detection in weighted graphs

We used a different set of datasets for this task since almost all of the datasets that were used in the previous algorithms were unweighted and singled edged. The following datasets were used. Though all the datasets are unweighted, weights are derived from the number of multiple edges between users.

Dataset	Category	Size	Edge Count	Description
Facebook ¹³	Social	47K	876K	facebook wall posts
Enron ¹⁴	Communication	87K	1.15M	user X sent mail to user Y
Digg ¹⁵	Communication	30K	87K	user X replied user Y

From Figure 15, we make the following observations.

1. In the **Enron** dataset, from the E vs N plot, we find outliers having a star anomaly. These users sent mail to a large number of other users that barely sent mail to each other. On the other hand, we see a large number of closely knit communities (cliques) of about 100 nodes. This is expected of this dataset since people in an organization from similar departments tend to know and send mail to each other.

From the W vs E plots, we find outlier users who are strongly connected to a single other user (outliers on the y-axis) and also some users having a heavy vicinity.

2. In the **Facebook** and **Digg** datasets, the E vs N plot do not provide outliers that significantly stands out although they tend towards the star configuration like the **Enron** dataset. The W vs E plot on the other hand reveals a lot of strongly connected users and user pairs.

¹³<http://konect.uni-koblenz.de/networks/facebook-wosn-wall>

¹⁴<http://konect.uni-koblenz.de/networks/youtube-u-growth>

¹⁵http://konect.uni-koblenz.de/networks/munmun_digg_reply

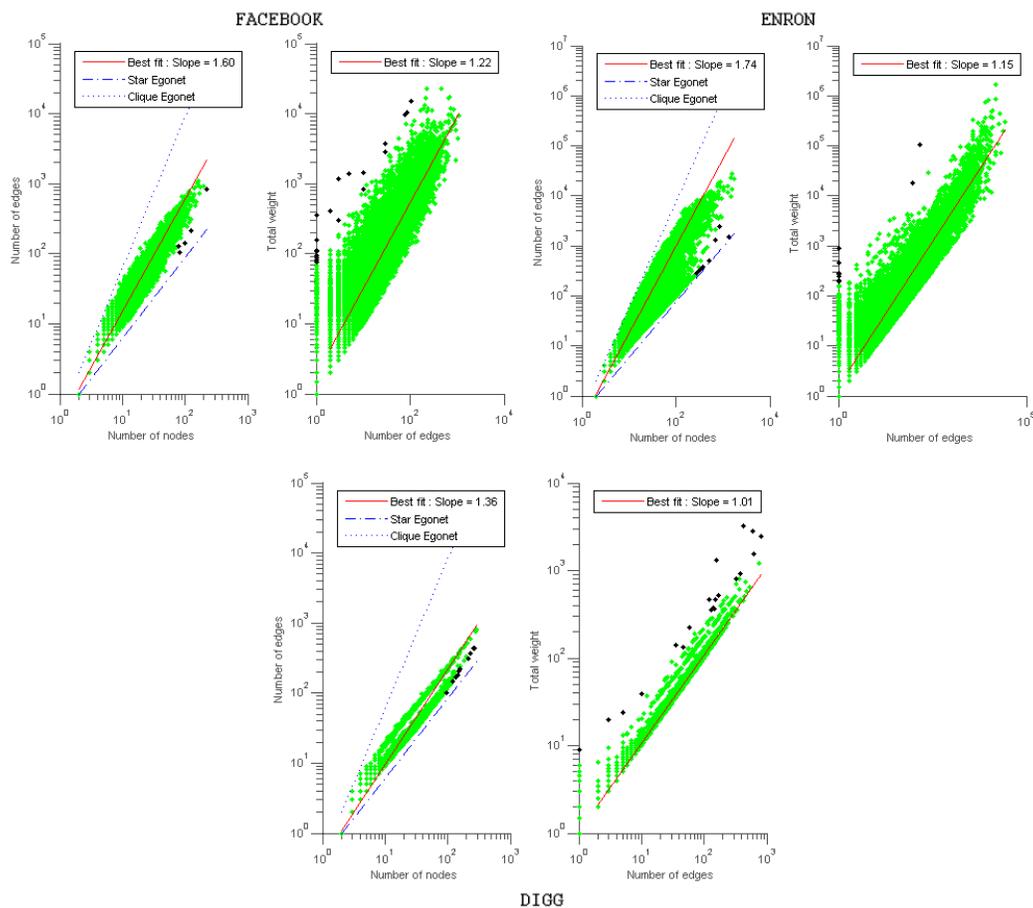


Figure 15: Anomaly Detection on Facebook, Enron and Digg datasets (From Top Left Clockwise). For each, plot of Number of Edges vs Number of Nodes and Total Weight vs Number of Edges are shown. The red line is the best fit line and characterizes the power law observed. The blue lines represent the Star Egonet (dashed) and Clique Egonet (dotted) configurations. The outliers are marked as black dots based on their outlier score.

5 Conclusions

Using SQL, we successfully implemented various graph mining algorithms and tested on a number of real world datasets. We found that SQL is quite powerful in accomplishing complex tasks using only a few lines of code. Several graph mining algorithms are based on matrix operations and these operations can be implemented very efficiently in SQL using joins.

We analyzed the patterns found in real world datasets and conclude that quite a lot of them obey power law distribution.

References

- [1] Leman Akoglu, Mary McGlohon, Christos Faloutsos: oddball: Spotting Anomalies in Weighted Graphs. PAKDD (2) 2010: 410-421
- [2] B. Aditya Prakash, Ashwin Sridharan, Mukund Seshadri, Sridhar Machiraju, Christos Faloutsos: EigenSpokes: Surprising Patterns and Scalable Community Chipping in Large Graphs. PAKDD (2) 2010: 435-448
- [3] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, Christos Faloutsos: gbase: an efficient analysis platform for large graphs. VLDB J. 21(5): 637-650 (2012)
- [4] U. Kang, Duen Horng Chau, Christos Faloutsos: Pegasus: Mining billion-scale graphs in the cloud. ICASSP 2012: 5341-5344
- [5] U. Kang, Duen Horng Chau, Christos Faloutsos: Mining large graphs: Algorithms, inference, and discoveries. ICDE 2011: 243-254
- [6] Danai Koutra, Tai-You Ke, U. Kang, Duen Horng Chau, Hsing-Kuo Kenneth Pao, Christos Faloutsos: Unifying Guilt-by-Association Approaches: Theorems and Fast Algorithms. ECML/PKDD (2) 2011: 245-260

A Appendix

A.1 Labor Division

The labor division on the various tasks.

Task	Attribution
Implementation of Framework	Nijith
Task 1: Degree Distribution	Nijith
Task 2: PageRank	Nijith
Task 3: Connected Components	Sharif
Task 4: Radius of every node	Sharif
Task 5: Eigenvalue/Singular Value	Nijith
Task 6: Belief Propagation	Sharif
Task 7: Count of Triangles	Nijith
Task 8: Broad Spectrum Mining	Nijith/Sharif
Extra: Anomaly Detection	Nijith/Sharif