# Concrete Graph Mining: a Database approach

*Wei Chen*
Dept. of Computer Science
CMU
weichen1@andrew.cmu.edu

*Siping Ji*
Dept. of Computer Science
CMU
sipingji@cmu.edu

November 30, 2013

**Abstract**

In this study, we focus on how to implement various graph mining algorithms based on RDBMS. Since the introduction of MapReduce, it has been considered the only silver bullet in large scale data analysis. While from database's perspective, RDBMS has been equipped with all the requierd capabilities for over twenty years. So why not RDBMS? We want to justify RDBMS's role in future big data application by experiment with current graph mining algorithms using RDBMS. The main contribution of this study is to explore the limitation in RDBMS's expressiveness and scalability, which scenario RDBMS will be a more appropriate alternative than MapReduce.

## 1 Introduction

The problem we want to solve is the following:

- GIVEN: a graph stored as edge list in RDBMS
- FIND: hidden patterns in this graph using various graph mining algorithms implemented by SQL language
- to MINIMIZE: the computational cost.

With the growth of computer network, graphs are ubiquitous nowadays, social networks, papers citing network, world wide web to a few. On the other hand, with the drastic drop of storage cost and the emergence of large social network companies like Facebook and Linkedin, graphs are now of a unprecedented size with billions of vertexes and edges. Mining such large dataset may help us gain lots of useful information and leads to interesting applications anomaly detection, social network analysis and so on. There are also lots of famous graph mining algorithms like PageRank, Random Walk with Restart, Belief Propagation who are aiming to find interesting patterns in graphs. However, most of these algorithms assume the graph fit in memory, which is apparently no longer suitable for the giant graphs today.

In this paper, we focus on how to use SQL language to implement efficient graph mining algorithms to find interesting patterns in large graphs that are stored in relational databases. We want to justify that RDBMS, equipped with highly optimized query engine and in-memory index, is sufficient for us to develop efficient and scalable graph mining algorithms in order to deal with graphs in this big data era.

# 2 Survey

Next we list the papers that each member read, along with their summary and critique.

## 2.1 Papers read by Siping Ji

The first paper was the "PEGASUS" paper by Kang [6]

- *Main idea*: This paper proposes a parallel programming primitive GIM-V that captures the shared characteristics among many seemingly different graph mining algorithms. The key idea is that the computation of graph mining algorithms like Pagerank, RWR, diameter estimation and so on, are essentially iterations of a generalized matrix-vector multiplication. More specifically, this generalized matrix-vector computation step can be captured by three basic operators in GIM-V - combine2, combineAll and assign. Based on this abstraction, many graph mining algorithms can be effciently computed via GIM-V over hadoop. Further, the author addresses the problem of high computational overhead of the shuffling stage in map-reduce. To overcome the problem and thus enhance the scalability, the author introduces several techniques, namely blocking, edge-clustering, diagonal block iteration and node renumbering into the implementation of GIM-V. Then the author conducts experiments showing that these optimization techniques siginificantly improves the scalability of the proposed method.
- *Use for our project*: Although GIM-V is a hadoop-based implementation, it is still a good reference for us since matrix-vector multiplication is easy to implement under SQL. We can also implement the operator primitives of GIM-V(combine2, combineAll and assign) using user-defined function in PostgreSQL.
- *Shortcomings*: This paper only compares the performance of GIM-V with different optimization techniques, but lack the comparison with other graph mining models or methods.

The second paper was "Patterns on the Connected Components of Terabyte-Scale Graphs" by Kang [4]

- *Main idea*: This paper studies and analyzes the patterns in connected components in real world large graphs. By introducing the concept of Graph Fractal Dimension(GFD) as a measure of the density of a connected component and the maximum effective radius(MER) vs. average effective radius(AER) ratio, the author shows there exists both consistency and difference among connected components of different size. Besides, the

author also suggests an log-linear relationship between rebel probability of a newcomer node and its degree in dynamically evolving graphs. Based on these observations, the author proposes a Community Connection model to explain the growth process of a graph and justifies that this model correctly captures the patterns they discovered.

- *Use for our project*: It is very relevant to the task of implementing the connected component algorithm. Additionally, this paper exemplifies how to discover, analyze and model the hidden pattern in a graph instead of treating statistics only as useless numbers. I think this helps us better explain the result in our experiment process.
- *Shortcomings*: Though this paper discusses extensively about analyzing and model patterns of connected components of a graph, it doesnt address much about how to apply this pattern and model to applications. Also, the implementation detail of finding connected components is ignored in the paper, which is what our project mainly focuses.

The third paper was "Inference of Beliefs on Billion-Scale Graphs" by Kang [3]

- *Main idea*: Belief Propagation(BP) is an popular graphical model algorithm for inferring the states of nodes in Markov Random Fields, it has been successfully applied to many problems in the fields of social network analysis, computer vision and so on. There already exists many efficient BP algorithms, but all of them assumes the graph can fit in main memory. This paper addresses exactly the problem of how to scale up the belief propagation(BP) algorithm to giant graphs with billions of nodes that can only stored on disks. The author first proposes the GIM-V - a primitive for parallel graph mining algorithms based on hadoop. It is based on the observation that the computation process of many existing graph mining algorithm like pagerank, random walk with restart are essentially repeated matrix vector multiplication by customizing the sub-operations in a matrix multiplication. Next, the author shows that by converting the original graph to a directed line graph, the key step in the BP computation that updates the messages vectors can be viewed as a process of matrix-vector multiplication, and thus can be applied to the framework of GIM-V. To further improve the efficiency of the parallel computing algorithm, the author introduces a trick of lazy multiplication to reduce the computation cost of multiplication operations. In the experiment, it shows that the method proposed beat the single machine BP algorithm in terms of running time, and when the number of machines increases, the algorithm scales up near linearly.
- *Use for our project*: Although the algorithm proposed in this paper is based on hadoop, it's still insightful to see the reinterpretation of the process of computing BP as a matrix-vector multiplication step. This insight may inspire us to efficiently compute BP in RDBMS, since the computation of sparse matrix-vector multiplication can be very efficient due to highly optimized query execution and in-memory index.
- *Shortcomings*: The major weakness of this paper is that it only compare their method to single-machine BP algorithm. It also lacks the theoretic justification for the nearliner scalability of their algorithm.

## 2.2 Papers read by Wei Chen

The first paper was "Mining Large Graphs: Algorithms, Inference, and Discoveries". [2]

- *Main idea*: This paper discuss how to do inference in graphical model under distributed setting, what if graph can not fit into main memory. They propose a variant of Belief Propagation called *Line Graph Fixed Point(LFP)* to address this issue. The key step is to induce a new graph called *Directed Line Graph* L(G) from original graph, it flips the definition of node and edge. Each node of L(G) represents an edge of G and edge exists between node $n_i$ and $n_j$ if and only if their related edges are incident. L(G) has a nice property that it's easy to derive exact recursive equation for it, actually the message updating equation of original Belief Propagation [15] can be adopted without modification. Then the author generalize mesage updating procedure as general Matrix-Vector operations, namely combine2(), combineAll(), sumVector() and assign() from [6]. While directly apply LFP without careful design will lead to drastic storage requirement to store L(G), the idea to tackle this is *Lazy Multiplication*, all operations on L(G) is is actually run on G. They propose an algorithm employing this idea, and run on Hadoop, its name is *Hadoop Line Graph Fixed Point*. Ha-LFP solves the biggest issue of the most graph mining algorithm — scalability. Since Ha-LFP is based on Hadoop, it inherits fault tolerance, data replication natively.
- *Use for our project*: LFP seems like a promising method to do Belief Propagation, combine2, combineAll, assign is natural in SQL. Matrix operation is tractable in SQL, and can be optimized by RBDMS. The performance could be comparable to Hadoop version. Also we don't need to construct L(G) explicitly, this avoids expensive cost in storage and time to access and update L(G).
- *Shortcomings*: It's kind of tricky to say this algorithm solve scalability perfectly. I think this is more contributed to Hadoop instead of LFP. There may exist better formulation of Belief Propagation on Hadoop. The experiments are conducted on M45, which is one of the most advanced supercomputer.

The second paper was "Understanding Belief Propagation and its Generalizations". [15]

- *Main idea*: Marginal Probability is important in graphical model inferencing, while most of the time it is expensive to calculate, the running time is usually exponential to the number of nodes. *Belief Propagation* is a neat technique to bring the running time down to linear to the number of edges in graphical model, and it has been widely used beyong machine learning, like computer vision[1], turbo code [11], etc. The success of Belief Propagation results from a property called *Conidtional Independence*. Most graphical models have the property that there exist many conditional independence between nodes. The core idea of Belief Propagation is to exploit this property to decouple the computation of global probability into several local computations. The marginal probability of node $n_i$ is called *belief*. Each node $n_i$ sends messages to other nodes $n_j$. *Message* $m_{ij}(n_j)$ represents how node $n_i$ thinks node $n_j$'s state should be. A node can calculate its belief once it has received messages from all its neighbouts. So

each edge is associate with two messages. The paper shows that each message can be calculated only once, thus we can use dynamic programming to compute every node's belief in linear time. The paper also introduce Generalized Belief Propagation which extends BP to group of nodes.

- *Use for our project*: This paper discusses how to compute belief of each node in one round(for trees). We can first implement the algorithms mentioned in this paper as a baseline. It is also helpful for understanding other optimizing algorithms for Belief Propagation.
- *Shortcomings*: There is not much discussion about the implementation in SQL. But user defined functions provided by PostgreSQL has limited expressiveness, so the real implementation will be quite different.

The third paper was "A Comparison of Approaches to Large-Scale Data Analysis". [13]

- *Main idea*: This paper compares the tradeoff between Parallel DBMS and MapReduce framework in large data analysis. The authors want to advocate the capability of Parallel DBMS in implementing large scale analytic tasks. The paper analyze the fundamental difference of Parallel DBMS and MapReduce, their support in schema, data distribution, fault tolerance, index, and utility tools. Then they conduct experiments about their performance in different tasks, like pattern searching, aggregation, join, etc. They conclude that Parallel DBMS outperforms MapReduce in all kinds of tasks. But Parallel DBMS requires a lot of effort to configure and profile to reach good performance, MapReduce is relatively much easier to use. In the end, the paper admit MapReduce's advantage over complex Parallel DBMS, while still suggest Parallel DBMS as an option for specific tasks.
- *Use for our project*: We can borrow some ideas from theirs experiments about investigate SQL function. Their analysis method is also a good model to follow.
- *Shortcomings*: The tasks they experimented are kind of trivial. We don't know the real difference in when the systems are used for more complex graph mining algorithms.

The fourth paper was "Pig Latin: A Not-So-Foreign Language for Data Processing". [12]

- *Main idea*: Either RDBMS or MapReduce framework represents some extreme in large scale data analysis, either unnatural for programmers' mind(SQL) or too low level to express many algorithms(MapReduce). Different solutions are proposed. Pig is a platform of Yahoo that try to reach a balance between these two styles to support more flexible(ad-hoc) data analysis tasks. It takes a hybrid strategy, provides friendly user interface(programming, debugging), while actual tasks are running on powerful Hadoop platform without loss of performance. And it propose a new high level programming language called *Pig Latin* that gives user more control of data flow, which programmer can adopt an imperative style, while retain the expressiveness of SQL. It also support several key operations that can be automatically parallelized, like filter, cogroup, group, join, etc. Its underlying implementation is compile Pig Latin programs into Hadoop jobs, so that it is also equipped with the benefits of MapReduce.

- *Use for our project*: Pig Latin's SQL-like syntax provides finer grained control over dataflow, which is more suitable for implementing graph mining algorithms. We can compare with SQL user defined function implementation of these algorithms, and analyze their intrinsic data accessing characteristic.
- *Shortcomings*: There are few graph mining algorithms implementation in Pig Latin, only a few open source analysis package like Linkedin's Datafu. And Pig only supports read-only data analysis workloads, which may lead to many unnecessary efforts if we want to do extensive write during computation.

The fifth paper was "Fast counting of triangles in large real networks without counting: Algorithms and laws" [14].

- *Main idea*: The paper proposed a analytical method to count the number of triangles in social networks. It is based upon two theorems regarding the fundamental property of adjencency matrix. The first theorem is that the number of global triangles is proportional to the sum of cubes of its eigenvalues, which is more formally defined as follow:

$$\Delta(G) = \frac{1}{6}\sum_{i=1}^{n}\lambda_i^3 \tag{1}$$

  Another one is about local triangle, the formula for its count is as follows:

$$\Delta_i = \frac{\sum_j u_{ij}^2 \lambda_j^3}{2} \tag{2}$$

  The author conducted several experiments on different datasets. and discovered several patterns using visualization, such as power law.
- *Use for our project*: For task 7, we adopt the algorithms described in the paper to count the number of global triangles and local triangles. The algorithm to calculate eigenvalues and eigenvectors is Lanczos, which is well suited for solving approximate top-k eigenvalues and eigenvectors.
- *Shortcomings*: The paper didn't discuss the selection of k, which is the number of top eigenvalues use for calculation. So most implementations have to randomly pick a small number and verify its effect. And how the magnitude of k will affect the running time of the algorithm.

# 3 Proposed Method

We implement all the graph mining algorithms using Postgres's embedded PL/pgSQL programming language, which supports many advanced features, like user defined function, aggregate, etc. It also has a sophisticated query execution engine, which we think is the most critical component of Postgres.

The following are the algorithms we plan to implement. The reason we choose them is that there are a lot of implementation in other platform like MapReduce, we can compare our SQL version with them to draw a clusion about SQL's unique charastic in solving data analytic tasks.

**Degree Distribution:** Plot the distribution of each node's degree.

**PageRank:** Determine the importance of every node of a graph based on its connectivity.

**Connected Components:** Partition the nodes of a graph also based on its connectivity.

**Radius of Every Node:** Compute the radius of every node in a graph. The radius is defined as the number of hops that a node needs to reach to its furthest neighbor.

**Belief Propagation:** Calculate the marginal probability of each node in a graphical model.

**Eigenvalue:** Using approximation method to estimate the top-k eigenvalues of a matrix.

**Count of triangle** Using approcimate method to calculate the number of triangles in a social network.

**Shortest Path** Calculate the shortest path from each node in a graph to a single source node.

**Minimum spanning tree** Construct a tree which is a subgraph of original graph with the minimum sum of weight of its edges.

## 3.1 Degree Distribution

We employ Postgres's group by command according to either source node or target node to count the degree distribution according to each node. The pseudo code is in Algorithm 1, 2.

**Algorithm 1** Out Degree distribution

```
Group edges according to source node's id
Count the number of members of each group
```

**Algorithm 2** In Degree distribution

```
Group edges according to target node's id
Count the number of members of each group
```

### 3.1.1 Math

First we need to count degree of each node. Then we count the frequency of each degree count.

### 3.1.2 Idea of SQL implementation

The only operation we need from SQL is its group by clause. We can aggregate the edges according with source node or target node.

### 3.1.3 SQL code

Please refer to the code in **Appendix**.

## 3.2 Pagerank

The algorithm of pagerank is Power method. We do matrix multiplication continuously until the change in pagerank is small. The most important equation for calculating pagerank is as follows:

$$PR(i) = \alpha \frac{1}{N} + (1 - \alpha) \sum_{j \in InNeighbor(i)} \frac{PR(j)}{OutDegree(j)} \tag{3}$$

**Algorithm 3** Pagerank

Bulk load graph into an edge table in database.
Initialization(damper factor=0.85, max iteration = 100, epsilon = 0.0001)
Build a weight matrix trans, initialize pagerank p
**repeat**
  For each node i, update its pagerank with its old value and its income node's pagerank.
**until** Convergence

### 3.2.1 Math

According to the definition of pagerank, we can gain an intuitive idea of how to calculate the pagerank of a node. We just take weighted sum of its incoming neighbors. We can encode this operation as matrix vector multiplication. We can do this multiple times. And the fix point of the equation is the pagerank of the graph. And by the results from linear algebra, we know that the stable vector is the eigenvector with biggest eigenvalue. Thus we can get the eigenvector by power method, which just do multiplication until convergence.

### 3.2.2 Idea of SQL implementation

The implementation consists of several components. First, we have a *graph* which has the schema *(from_id, to_id, weight)*. Then we will build a *rank* table has the schema *(node_id, rank)*, all the rank are initialized randomly. After we enter loop, we will do a large matrix vector multiplication, which is implemented by SQL *select* and *join*. The new pagerank is stored in a temporary table. After the loop is over, we calclate the updates to every node, if the change is little, then abort.

### 3.2.3 SQL code

Please refer to the code in **Appendix**.

## 3.3 Weakly connected components

In terms of the implementation of weakly connected components. We borrow the idea of HCC method from the "PEGASUS" paper.[6]

### 3.3.1 method

The key idea of this algorithm is that for every node $v_i$ in the graph, we maintain a component id $c_i^h$ which is the minimum node id within $h$ hops from $v_i$. Initially, $c_i^h$ of $v_i$ is set to its own node id. For each iteration, each node sends its current component id to its neighbors. Then $c_i^{h+1}$ is set to the minimum value among its current component id and the received component ids from its neighbors. Finally, when the update converges, all nodes in the same connected component will share the same component id.

### 3.3.2 Idea of SQL implementation

The algorithm can be described in algorithm 4. The key step that updates the component id to the minimum of its neighbors' is accomplished in SQL using join and group by clause. After several rounds of iteration, the nodes in the same connected component will share the

**Algorithm 4** Weakly Connected Component

Bulk load graph into an edge table in database.
Create a component table, where each entry contains a node id, and the component id.
Initialize the component table where component id equals node id.
**repeat**
  For each node, assign the minimum component id of its neighbors as the new component id of this node.
**until** Convergence

same component id. The number of iterations for convergence can be proved to be upper bounded by the diameter of the graph.

### 3.3.3 SQL code

Please refer to the code in **Appendix**.

## 3.4 Radius of every node

We discard the traditional algorithm because it is extremely infeasible for large graphs, since it uses a set to record every neghbors within n hops for a node during iteration, which requires a $O(n^2)$ space.

### 3.4.1 method

Since the exact algorithm is hopeless, we use the approximation algorithm described in "HADI" paper[7] instead. Specifically, we use the Flajolet-Martin algorithm for counting the number of distinct members in a multiset. It is guaranteed to give an unbiased estimate and a tight O($log(N)$) bound for space complexity. The basic idea of Floajolet-Martin algorthm is to use a bitstrings of length $L$ to encode the set. For each element to add, we randomly pick up a index according to a specified distribution, and assign BITMAP[index] to 1. Following this procedure to add element, the size of the final set can be estimated by $\frac{1}{\phi}2^{\frac{1}{k}\sum_{i=1}^{k}R_i}$, where $\phi = 0.77351$, $R_i$ denotes the index of the leftmost 0 in the the $k$th bitstring.

In our proposed method for computing radius of every node, we use the Flajolet-Martin(FM) bitstrings to encode the neighbors of every node. Formally, we use $k$ FM-bitstrings $b(h,i)$ to represent the set of neighbor nodes reachable from $node_i$ within h hops. And for each iteration, we use the following way to update each FM-bitstring:

$$b(h,i) = b(h-1,i) \quad BIT-OR \quad b(h-1,j)|(i,j) \in E$$

Given the above description of how to encode neighbors of a node, and the method to update bitstrings, we can describe the approximation method we used to compute the raidus of every node in algorithm 5.

**Algorithm 5** Radius of Every Node

Bulk load graph into an edge table in database.
Preprocess the edge table, add a self loop edge to every node in the graph.
Initialize the vertex table, which contains node id, and a column of bitstring array, the bitstrings are initialized using the FM algorithm
**repeat**
  For every node update the bitstring according to formula: $b(h,i) = b(h-1,i) \quad BIT-OR \quad b(h-1,j)|(i,j) \in E$.
  For every node, check whether the bitstrings is unchanged before and after updates, if it's not changed, output i as the radius for this node.
**until** The bitstrings of every node stabilizes or it reaches the maximum rounds of iteration.

### 3.4.2 Idea of SQL implementation

In order to store the fm-string array in the table, we use the array type which is supported by PostgreSQL. To initialize the fm string and update the fm string array, we defined some user defined functions in PostgreSQL. The key step of algorithm that updates the FM-bitstring array is accomplished by using aggBitOr in the join and group by clause. Specifically, we join the edge table and vertex table on dst id, group by src id, and then call the aggBitOr to update the fm string array for all nodes. We summarize the user defined functions in Table 1.

Table 1: User defined functions for task 4

| function name | description |
|---|---|
| fmAssign | Assign the k FM-bitstrings for a node |
| bit-or | Execute the OR operations between two bitstring arrays |
| aggBitOr | Aggregate function for bit-or |
| fmSize | Estimate the size of a set encoded by FM-bitstring |

### 3.4.3 SQL code

Please refer to the code in **Appendix**.

## 3.5 Eigenvalue

We adopt the method propose in [5]. There are several methods to solve part of eigenvalue computation problems, for instance, power method[10]. While it has the limitation that it can only extract the eigenvector with biggest eigenvalue. Several method have been proposed to extract top k eigenvectors simultaneously. The approach we use is Lanczos algorithm[9]. The general idea about this algorithm is that instead of directly work on an $N \times N$ matrix, we first generate a skinny $N \times m$ matrix(M $\ll$ N). Then it computes a small $M \times M$ dense matrix which has good approximation to the eigenvalues of the original matrix. In this case, we directly apply quadratic algorithm to top-k eigenvalues. Notice that k $<$ M.

**Algorithm 6** Lanczos algorithm

**Input:** Matrix $A^{n \times m}$
random n-vector $b$,
number of steps m
**output:** Orthogonal matrix $V_m^{v \times m} = [v_1 \cdots vm]$,
coefficients $\alpha[1..m]$ and $\beta[1..m-1]$

1: $\beta_0 \leftarrow 0, v_0 \leftarrow 0, v_1 \leftarrow \frac{b}{\|b\|}$
2: **for** $i = 1$ to $m$ **do**
3: $v \leftarrow Av_i$
4: $\alpha_i \leftarrow v_i^T v$
5: $v \leftarrow v - \beta_{i-1} v_{i-1} - \alpha_i v_i$
6: $\beta_i \leftarrow \| v \|$
7: **if** $\beta_i = 0$ **then**
8: break for loop
9: **end if**
10: $v_{v+1} \leftarrow \frac{v}{\beta_i}$
11: **end for**

**Algorithm 7** Build tridiagonal matrix

**Input:** $\alpha, \beta$ **Output:** $T_m^{m\times m}$

1: **for** $i = 1$ to m **do**
2: $\quad T[i,i] \leftarrow \alpha_i$
3: $\quad T[i,i+1] = T[i+1,i] \leftarrow \beta_i$
4: **end for**

**Algorithm 8** Compute Ritz values

**Input:**Orthogonal matrix $V_m^{n\times m}$
coefficients $\alpha[1..m]$ and $\beta[1..m-1]$

1: $T_m \leftarrow$ (build a tridiagonal matrix from $\alpha$ and $\beta$)
2: $QDQ^T \leftarrow EIG(T_m)$
3: $\lambda_{1..k} \leftarrow$ (top k eigenvalues from D)
4: $Q_k \leftarrow$ (k columns of Q corresponding to $\lambda_{1..k}$)
5: $R_k \leftarrow V_m Q_k$

### 3.5.1 Math

Different from power method, the intermediate multiplication matrix is used to construct a set of orthonormal base of *Krylv subspace* $K_m$ which follows the definition:

$$K_m =< b, Ab, \cdots, A^{m-1}b > . \tag{4}$$

The sub procedure to construct orthonormal bases may be any standard algorithm, for example Gram-schmidt algorithm. We can view Lanczos algorithm as an iterative method which incrementally construct Krylov subspace. The pseudo-code is shown in Algorithm 3.5.

After Lanczos factorization, we get a few matrices that satisfy the following equation:

$$AV_m = V_m T_m + f_m e_m^T \tag{5}$$

To name a few, $A^{n\times m}$ is input matrix, $V_m^{n\times m}$ contains the m orthonormal bases, $T_m^{m\times m}$ is a tridiagonal matrix, $f_m$ is new n-vector orthogonal to all columns of $V_m$, $e_m$ is a vector that $m$th element is 1, and others 0. After algorithm 3.5, we need to construct the matrix $T_m^{m\times m}$. The algorithm is quite simple, it is listed in algorithm 7.

The eigenvalues of $T_m$ are called Ritz values, and $V_m Y$'s columns are called Ritz vector. It is constructed by Algorithm 8. We expect the Ritz values and Ritz vectors to be good approximation of the eigenvalues and eigenvectors of original matrix. The computation of eigenvalues of $T_m$ can be done by standard quadratic algorithms, such as QR method.

### 3.5.2 Idea of SQL implementation

Since the algorithm of Lanczos algorithm is matrix calculation intensive, so we wrap all matrix related operation in our host language Python. I'll list the most important routines

that appears very often in my high level implementation of Lanczos. Then in the final python code, I'l just call these wrappers instead of using raw SQL again and again.

**create_vector_or_matrix:** declare a vector/matrix variable

**assign_to:** Copy a variable's value to another variable

**vetorr_length:** Return the length of a vector

**vector_dot_product:** take the dot product of two vectors

**reverse_matrix:** Append reverse of every edge into original graph

**matrix_multiply_matrix_overwrite:** Multiply a matrix with a matrix

**matrix_multiplt_vector_overwrite:** Multiply a matrix with a vector

**normalzed_vector:** Normalize a vector

### 3.5.3 SQL code

Please refer to the code in **Appendix**.

## 3.6 Belief Propagation

For belief propagation, we use the fabp method proposed in paper[8].

### 3.6.1 method

It can be shown that the solution of belief propagation can be approximated by the linear system:

$$[\mathbf{I} + a\mathbf{D} - c\mathbf{A}]\mathbf{b_h} = \phi_\mathbf{h}$$

where $\mathbf{A}$ is the n by n symmetric adjacency matrix, $\mathbf{D}$ is the diagonal matrix of degrees, $b_h$ corresponds to the vector of final beliefs for each node, $\phi_h$ is prior belief vector, and $h_h$ is the homophily factor, $a = 4h_h^2/(1 - h_h^2)$ and $c = 2h_h/(1 - 4h_h^2)$.

To solve this linear system, we can see :$\mathbf{I} + a\mathbf{D} - c\mathbf{A}$ as the form $\mathbf{I} - \mathbf{W}$, where $\mathbf{W} = -a\mathbf{D} + c\mathbf{A}$, and using the expansion:

$$(\mathbf{I} - \mathbf{W})^{-1} = \mathbf{I} + \mathbf{W} + \mathbf{W}^2 + \mathbf{W}^3 + ...$$

and the solution of the linear system is given by the formula:

$$\mathbf{b_h} = (\mathbf{I} - \mathbf{W}^{-1})\phi_\mathbf{h} = \phi_\mathbf{h} + \phi_\mathbf{h}\mathbf{W} + \phi_\mathbf{h}\mathbf{W}^2 + \phi_\mathbf{h}\mathbf{W}^3 + ...$$

Given this power method, the implementation is pretty straightforward as described in algorithm 9.

**Algorithm 9** Belief Propagation

Bulk load graph into an edge table in database.
Initialize $h_h = 0.001$
Initialize the initial belief of every node as prior belief
**repeat**
  Update the belief of node by $b_h(i) = b_h(i-1)\mathbf{W} + \phi_\mathbf{h}$
**until** Convergence

### 3.6.2 Idea of SQL Implementation

The major computation involved is matrix vector multiplication, which is easy to implement in SQL using join and group by step. Furthermore we've wrapped all the matrix related operation in our host language Python, as described in previous task.

### 3.6.3 SQL code

Please refer to the code in **Appendix**.

## 3.7 Count of Triangle

We use a simple technique proposed by [14], its general idea is build upon a theorem that the count of triangles in a graph is proportional to the sum of cubes of eigenvalues of the graph.

### 3.7.1 Global triangle

The algorithm to calculate global triangles is as follows:

**Math** The formula to count global triangle is sum of cubes of eigenvalues, which is:

$$\Delta(G) \leftarrow \frac{1}{6}\sum_{j=1}^{i-1}\lambda_j^3 \tag{6}$$

### 3.7.2 Local triangle

The algorithm to calculate the local triangle is as follows.

**Math** $\Delta_i$ is the number of local triangles that node i participated in. The formula of local triangle count is based on the following theorem:

$$\Delta_j = \frac{\sum_{k=1}^{i-1} u_{jk}^2 \lambda_k^3}{2} \tag{7}$$

**Algorithm 10** The EigenTriangle algorithm

**Require:** Adjacency matrix A (n X n)
**Require:** Tolerance *tol*
**Output:** $\triangle'(G)$ global triangle estimation

$\lambda_i \leftarrow LanczosMethod(A, 1)$
$\overrightarrow{\Lambda} \leftarrow [\lambda_1]$
$i \leftarrow 2$ { initialize i, $\overrightarrow{\Lambda}$ }
**repeat**
  $\lambda_i \leftarrow LanczosMethod(A, i)$
  $\overrightarrow{\Lambda} \leftarrow [\overrightarrow{\Lambda}\lambda_i]$
  $i \leftarrow i + 1$
**until** $0 \leq \frac{|\lambda_i^3|}{\sum_{j=1}^{i} |\lambda_i|^3} \leq$ tol
$\triangle'(G) \leftarrow \frac{1}{6}\sum_{j=1}^{i} \lambda_i^3$
**return** $\triangle'(G)$

**Algorithm 11** The local eigentriangle algorithm[14]

**Require:** Adjacency matrix $A(n \times n)$
**Require:** Tolerance *tol*
**OUTPUT:** $\Delta'(G)$ per node triangle estimation
$\langle \lambda_1, \vec{u_1} \rangle \leftarrow LanczosMethod(A, 1)$
$\vec{\Lambda} \leftarrow [\lambda_1]$
$\bigcup \leftarrow [\vec{u_1}]$
$i \leftarrow 2$
**repeat**
  $\langle \lambda_i, \vec{u_i} \rangle \leftarrow LanczosMethod(A, i)$
  $\vec{\Lambda} \leftarrow [\vec{\Lambda}\lambda_i]$
  $\bigcup \leftarrow [\bigcup \vec{u_1}]$
  $i \leftarrow i + 1$
**until** $0 \leq \frac{|\lambda_i^3|}{\Sigma_{j=1}^{i-1}\lambda_j^3} \leq tol$
**for** $j = 1$ to $n$ **do**
  $\triangle_j = \frac{\Sigma_{k=1}^{i-1} u_{jk}^2 \lambda_k^3}{2}$
**end for**
$\triangle(G) \leftarrow [\triangle_1, \ldots, \triangle_n]$
**return** $\triangle(G)$

### 3.7.3 Idea of SQL implementation

We will call Lanczos to get the eigenvalues and eigenvectors of the graph, and sum up the number using SQL *select*.

### 3.7.4 SQL code

Please refer to the code in **Appendix**.

## 3.8 Shortest Path

We adopt Dijkstra's shortest algorithm to compute shortest path from the source node. It works for directed weighted graph which has $O(|V|log|V|+|E|)$ time complexity. The pseudo code is listed in algorithm 3.8.1.

### 3.8.1 Math

The core idea of Dijkstra's idea is that it greedily select a candidate node which has already obtained its minimum distance, then update its neighbors' current best distance.

**Algorithm 12** Dijkstra shortest path algorithm

**Input:** Source node and directed weighted graph.
**Output:** Shortest path of every node from source node.

```
for all node V in graph do
  dist[V] ← infinity
  visited[V] ← false
end for
dist[source] ← 0
insert source into Q
while Q is not empty do
  u ← vertex in Q with smallest distance
  remove u from Q
  visited[u] ← true
  for all neighbour v of u do
    alt ← dist[u] + dist_between(u, v)
    if alt < dist[v] && !visited[v] then
      dist[v] ← alt
      insert v into Q
    end if
  end for
end while
```

### 3.8.2 Idea of SQL implementation

This is implemented purely in SQL. Just like ordinary C code, I have a table to record the current status(visited, distance) of each node, and update one node at a time.

### 3.8.3 SQL code

Please refer to the code in **Appendix**.

## 3.9 Minimum Spanning Tree

We use the classical Prim's algorithm for this additional task. We abandon the Kruskal's algorithm because the disjoint set is not easy to implement using SQL.

### 3.9.1 method

The general idea of Prim's algorithm is to first initialize a tree with a single vertex, chosen arbitrarily from the graph, Then we grow the tree by one edge, the one that connect the tree to vertices not yet in the tree with minimum cost(weight). We repeat this process until all of the nodes in a graph is in the tree(of course we're assuming that the graph is weighted and connected).

### 3.9.2 Idea of SQL implementation

The implementation in SQL can be described in algorithm 3.9.2.

**Algorithm 13** Prim's algorithm

**Input:** Edge Table E of a undirected connected graph
**output:** Edge Table MST containing edges of the minimum spanning tree

```
Create a node table N
Randomly insert a node into N
for i = 1 to numberofnodes − 1 do
    Insert into MST an edge from E with minimum weight where src node is in N and
    destination node is not in N
    Insert into N with the destination node of the edge selected in last step
end for
```

### 3.9.3 SQL code

Please refer to the code in **Appendix**.

# 4 Experiments

## 4.1 Dataset

We have conducted experiments on the following datasets(Mainly For Phase 1):

| Dataset Name | Advogato |
|---|---|
| Largest conn compo | 5054 |
| Size | 6551 vertices |
| Volume | 51332 edges |

In out final experiment suit, a more large and diverse datasets from different domains will be explored. The tentative source of experiment dataset are listed below:

**SNAP:**It has abundant data about social network, we plan to conduct triangle counting, pagerank, radius computing experiment which will reveal the underlying feature of large graphs and spot strange graphs.

**Konect:**Konect has more diverse datasets compared to SNAP, like citation network. It's a good target to analyze features of non-social networks, we will examine whether such networks follow power law by generating degree distribution, etc.

We also calculated statistics about the weakly connected components in table **??**.We compute radius for every node in Advogato dataset, the result is in table 1:

## 4.2 Task 1: Degree distribution

### 4.2.1 Description

We conduct 5 experiments on large scale graph data(all with more than 1 million nodes). The details of each dataset is as follows:

| name | nodes | edges |
|---|---|---|
| Roadnet-ca | 1,965,206 | 5,533,214 |
| Roadnet-PA | 1,088,092 | 3,083,796 |
| Roadnet-TX | 1,379,917 | 3,843,320 |
| wiki-Talk | 2,394,385 | 5,021,410 |
| Youtube | 1,134,890 | 2,987,624 |

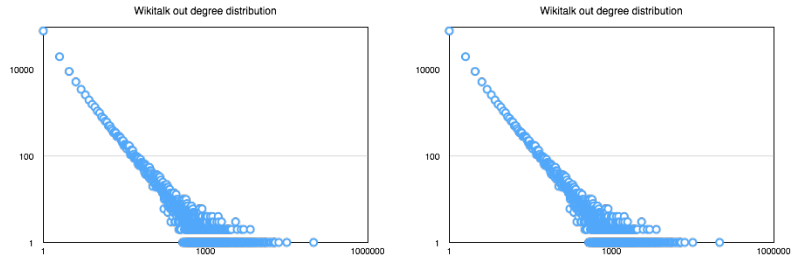
### 4.2.2 Detailed Plots

Following are the rank-frequency plots of each dataset, (a)(b) shows the in degree and out degree out Wikitalk. (c) is Roadnet-Ca. (d) is Roadnet-PA. (e) is Roadnet-TX. (f) is Youtube.

**Proof of Correctness:** For each dataset, we compare the plot with its original ground truth plot if it has.[1]. They are almost the same.

[1]Stanford SNAP does not have degree distribution, KONECT has degress distribution

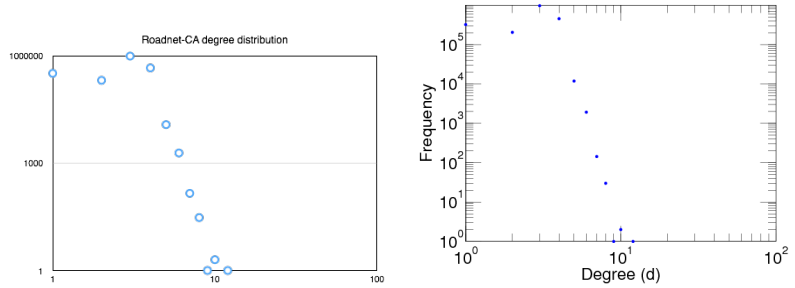
**Wiki-Talk** In the wiki-talk graph, it's directed. Each node represents a user, and an edge represents a user at least edited another user's talk page. Figure 1 shows the in and out degree of this dataset. We don't have official degree distribution plot for this dataset. We can observe **power law** from this plot.




(a) (b)

Figure 1: In degree distribution (a) and out degree distribution (b) of Wikitalk.

**Roadnet-CA** Roadnet-CA is un undirected graph. Each node represents an intersection of road and edge is the road segments between two road intersections. Figure 2(a) shows our plot of degree distribution, Figure 2(b) is the original degree distribution of the dataset. We can see that they are the same. There is no obvious power law.

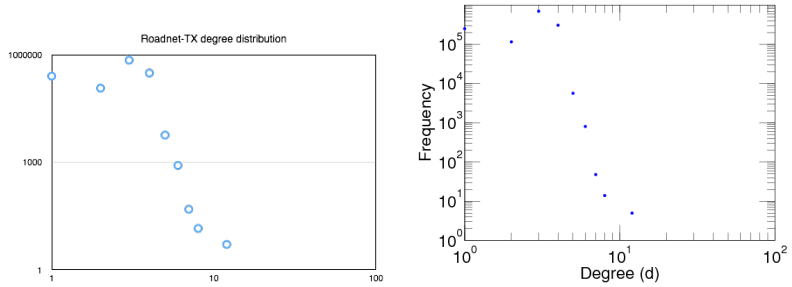



(a) (b)

Figure 2: (a)Roadnet-CA (b) official degree distribution of Roadnet-CA

**Roadnet-TX** Roadnet-TX is an undirected graph. Each node represents an intersection of road and edge is the road segments between two road intersections. Figure 3(a) shows our plot of degree distribution, Figure 3(b) is the original degree distribution of the dataset. We can see that they are the same. There is no obvious power law.

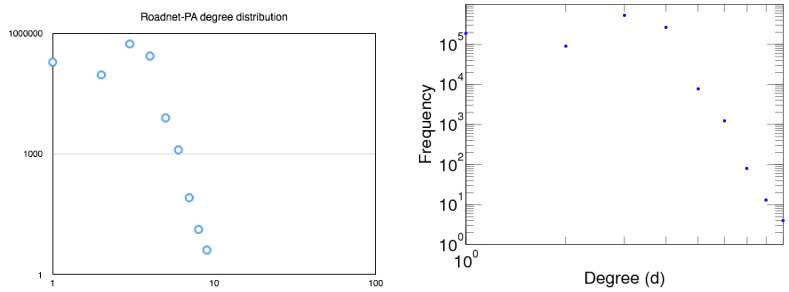
**Roadnet-PA** Roadnet-pa is un undirected graph. Each node represents an intersection of road and edge is the road segments between two road intersections. Figure 4(a) shows our




(a) (b)

Figure 3: (a)Roadnet-TX (b) official degree distribution of Roadnet-TX

plot of degree distribution, Figure 4(b) is the original degree distribution of the dataset. We can see that they are the same. There is no obvious power law.



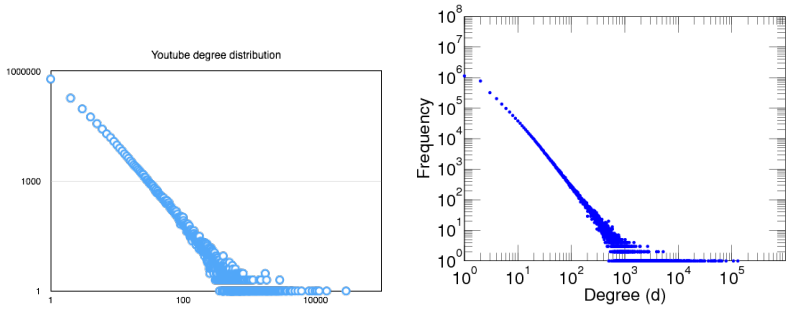

(a) (b)

Figure 4: (a)Roadnet-pa (b) official degree distribution of Roadnet-pa

**Youtube** Youtube is an undirected graph, each node is a user in youtube.com, an edge represents they are friends. Figure 5(a) shows our plot of the dataset, Figure 5(b) is the official degree distribution of the dataset. We can see that they are exactly the same and there exhibits **power law**.

### 4.2.3 Observation

As we can observe, that most *social network* exhibit perfect power law in degree distribution. It aligned with our intuition. While for the series of Roadnet dataset, we can not observe obvious power law. So maybe we can conclude that not all graphs have power law property in it. Some datasets like roadnet, involves a lot of human design, is less *chaotic* than ordinary graphs.




(a) (b)

Figure 5: (a)Youtube (b) official degree distribution of Youtube

## 4.3 Task 2: Pagerank

In this experiment, we run pagerank on various size of graph data, 10 in total. The purpose of this design is that we think pagerank is a time consuming operation, and our first try proves our intuition. So we want to test the implementation on incrementally larger dataset, so that we can spot the bottleneck of single machine implementation.

| Name | Nodes | edges |
|---|---|---|
| Roadnet-ca | 1,965,206 | 5,533,214 |
| Roadnet-PA | 1,088,092 | 3,083,796 |
| Roadnet-TX | 1,379,917 | 3,843,320 |
| com-Amazon | 334,863 | 925,872 |
| web-BerkStan | 685,230 | 7,600,595 |
| email-Enron | 36,692 | 367,662 |
| web-Google | 875,713 | 5,105,039 |
| soc-Slashdot0902 | 77,360 | 905,468 |
| web-Stanford | 281,903 | 2,312,497 |
| wiki-Vote | 7,115 | 103,689 |

### 4.3.1 Detailed Plot

**Proof of Correctness:** In order to verify the correctness of our implementation, we compare the SQL implementation with a standard implementation with NumPy[2] on a tiny dataset from "Introduction to Information Retrieval". Table 2 is the outcome of two different implementaions. We can see that most of the values are the same to one digit of the right of decimal point. The difference after that is due to subtle difference in implmentation, such as max iteration, initilization, etc.

Here We use rank-frequency plot to find the underlying patterns in the datasets.

[2]http://www.numpy.org/

| Node | SQL | NumPy |
|---|---|---|
| 1 | 0.122864 | 0.11799887 |
| 2 | 0.081234 | 0.08069813 |
| 3 | 0.263843 | 0.2526222 |
| 4 | 0.529185 | 0.52647168 |
| 5 | 0.452971 | 0.45487423 |
| 6 | 0.081234 | 0.08069813 |
| 7 | 0.645657 | 0.65203598 |

Table 2: SQL implementaion VS NumPy implementaion

**Roadnet-CA** The rank-frequency plot(Figure 6) of Roadnet-CA is nearly flat, there is not obvious pattern in the plot.

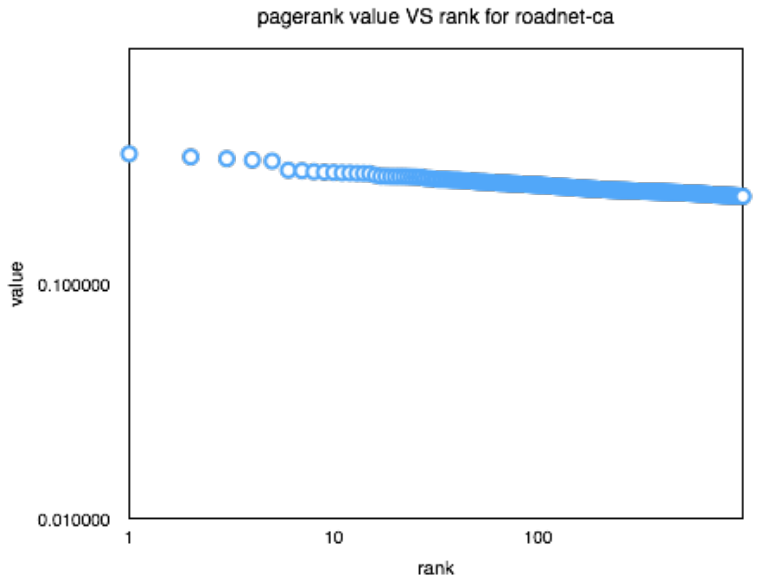



Figure 6: Rank-frequency plot Roadnet-CA

**Roadnet-TX** The rank-frequency plot(Figure 7) of Roadnet-TX is nearly flat, there is not obvious pattern in the plot.

**Roadnet-PA** The rank-frequency plot(Figure 8) of Roadnet-PA is nearly flat, there is not obvious pattern in the plot.

**com-Amazon** From the plot(Figure 9) we can find that the pagerank distribution follows **power law**, which means that top few nodes has the largest pagerank, while the majority

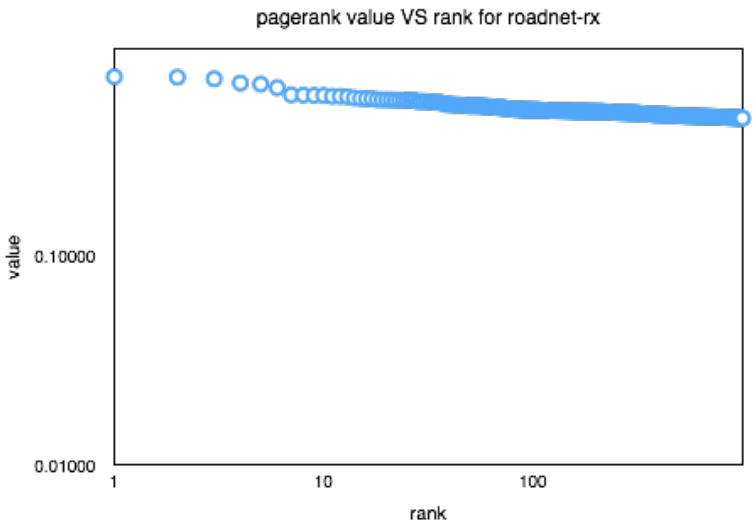



Figure 7: Rank-frequency plot Roadnet-TX

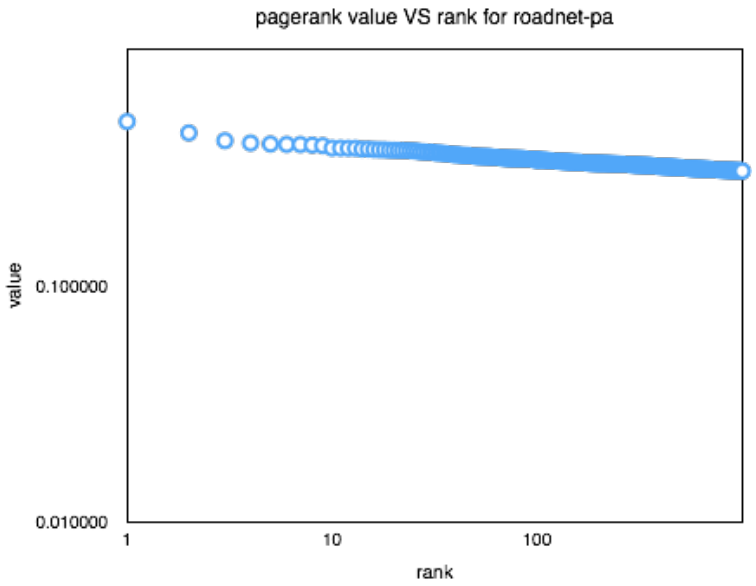



Figure 8: Rank-frequency plot Roadnet-PA

of the node has small pagerank. This matches out intuition in real world data.

**web-BerkStan** From the plot(Figure 10) we can find that the pagerank distribution follows **power law**(exception the first few nodes), which means that top few nodes has the largest pagerank, while the majority of the node has small pagerank. This matches out intuition in real world data.

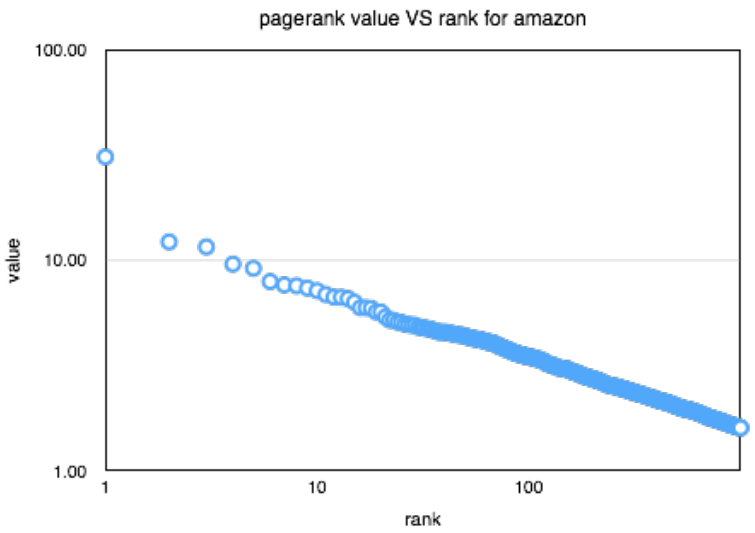



Figure 9: Rank-frequency plot Amazon

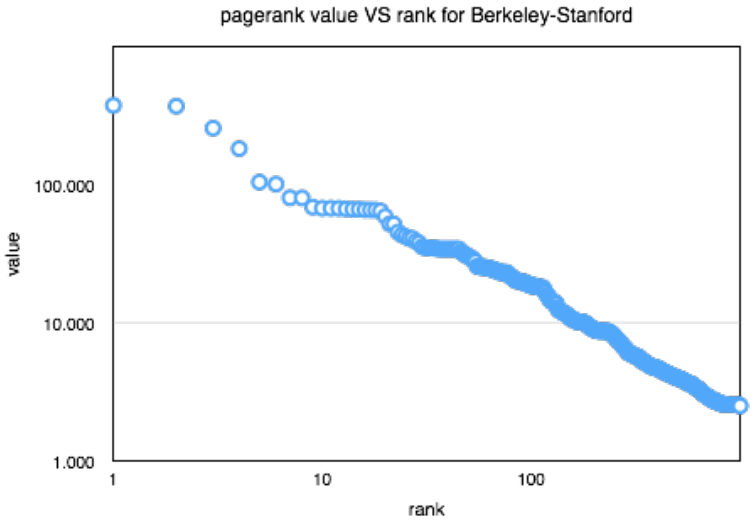



Figure 10: Rank-frequency plot BerkeStan

**email-Enron** From the plot(Figure 11) we can find that the pagerank distribution follows **power law**(except the first few nodes), which means that top few nodes has the largest pagerank, while the majority of the node has small pagerank. This matches out intuition in real world data.

**web-Google** From the plot(Figure 12) we can find that the pagerank distribution follows **power law**, which means that top few nodes has the largest pagerank, while the majority of the node has small pagerank. This matches out intuition in real world data.

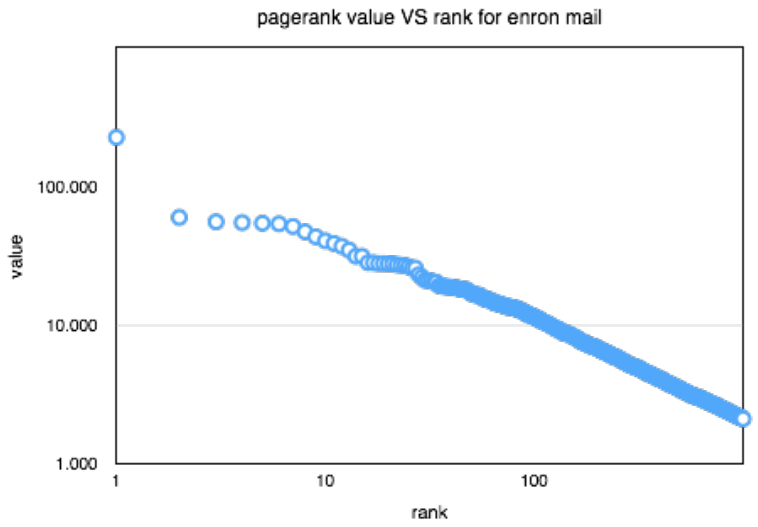



Figure 11: Rank-frequency plot Enron mail

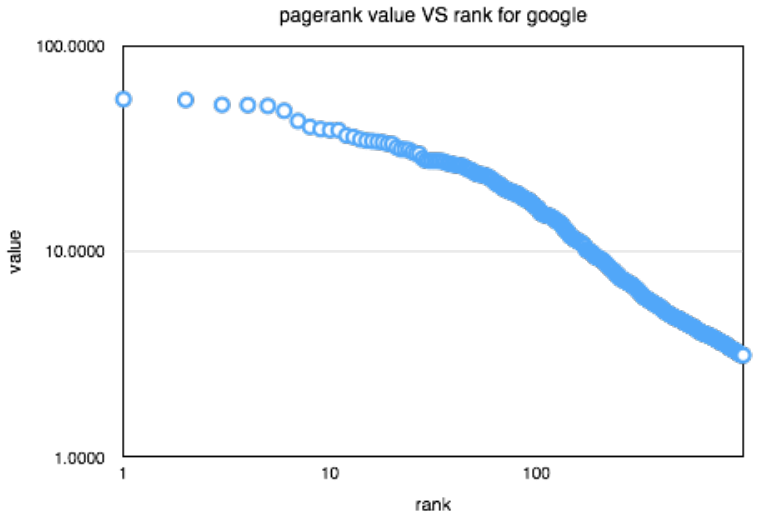



Figure 12: Rank-frequency plot Google

**soc-Slashdot0902** From the plot(Figure 13) we can find that the pagerank distribution follows **power law**, which means that top few nodes has the largest pagerank, while the majority of the node has small pagerank. This matches out intuition in real world data.

**web-Stanford** From the plot(Figure 14) we can find that the pagerank distribution follows **power law**(except the first few nodes), which means that top few nodes has the largest pagerank, while the majority of the node has small pagerank. This matches out intuition in real world data.

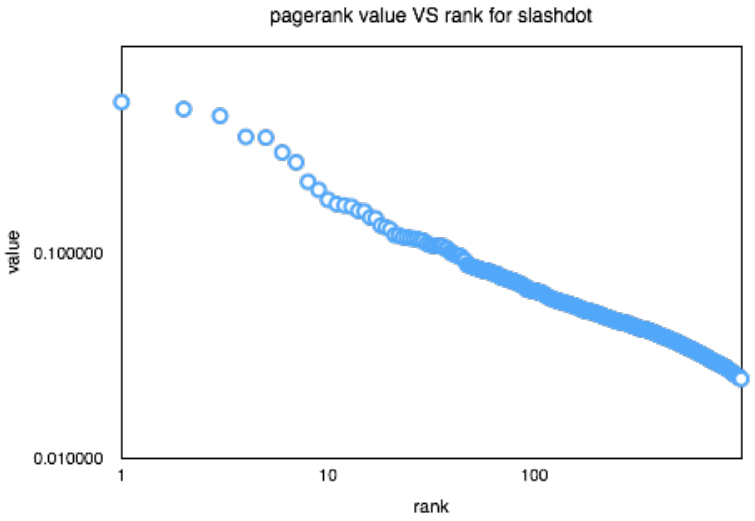



Figure 13: Rank-frequency plot Slashdot

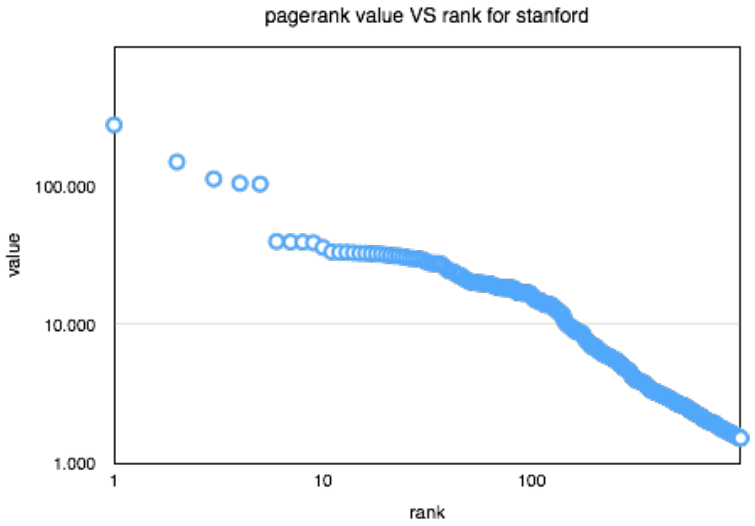



Figure 14: Rank-frequency plot Stanford

**wiki-Vote** From the plot(Figure 15) we can find that the pagerank distribution follows **power law**, which means that top few nodes has the largest pagerank, while the majority of the node has small pagerank. This matches out intuition in real world data.

### 4.3.2 Observation

We can observe that in log-log scale, all datasets exhibits linear relationship between rank and frequency. Thus we find another appearance of **power law** in natural graph. An abnormal phenomenon is that the slope of Roadnet datasets is nearly flat. This reflects

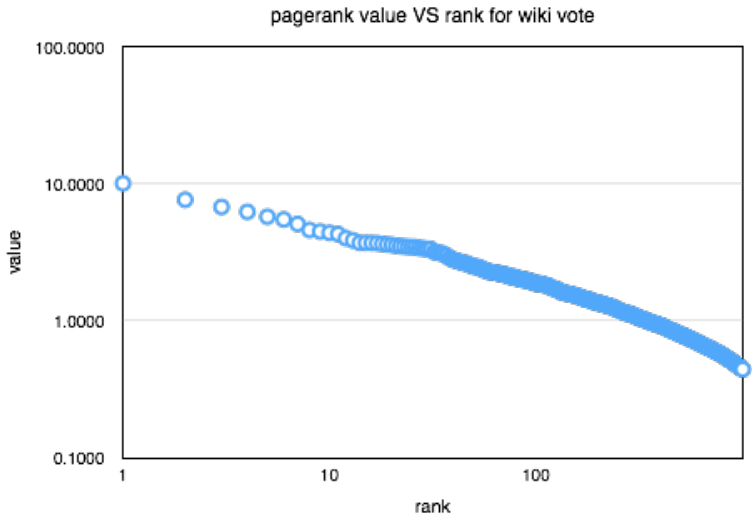



Figure 15: Rank-frequency plot Wiki-Vite

another aspects of the fundamental property of Roadnet, that all nodes in a roadnet graph are nearly equal to each other, it's a decentralized graph. While in ordinary social network, we always expect to have some important authorities.

## 4.4 Task 3: Weakly Connected Component

### 4.4.1 Validity

We verify the validity of our algorithm in both small and large datasets. For small datasets, we generate several synthetic graph containing a small amount of weakly connected components of different sizes. Then we run our algorithm on this small datasets, the result matches the reality. For large datasets, we use real graph from Konect and SNAP project, we can reproduce the statistics "number of nodes in largest WCC" on their web using our algorithm, which partially demonstrate the correctness of our algorithm .

### 4.4.2 Experiment on large datasets

In this experiment, we run weakly connected component in several datasets of various size (from 100 thousand to 2 million of nodes). In the following, we show the frequency radius plot in log log scale for dataset Email-EuALL (figure 16), Soc-Sign-Epinions(figure 17,) Trec-wt10g (figure 18), Google Web Graph (figure 19), Wiki Talk (figure 20).

We also run our algorithm in two large connected graphs(com-Youtube and com-amazon), some statistics are presented in table 3

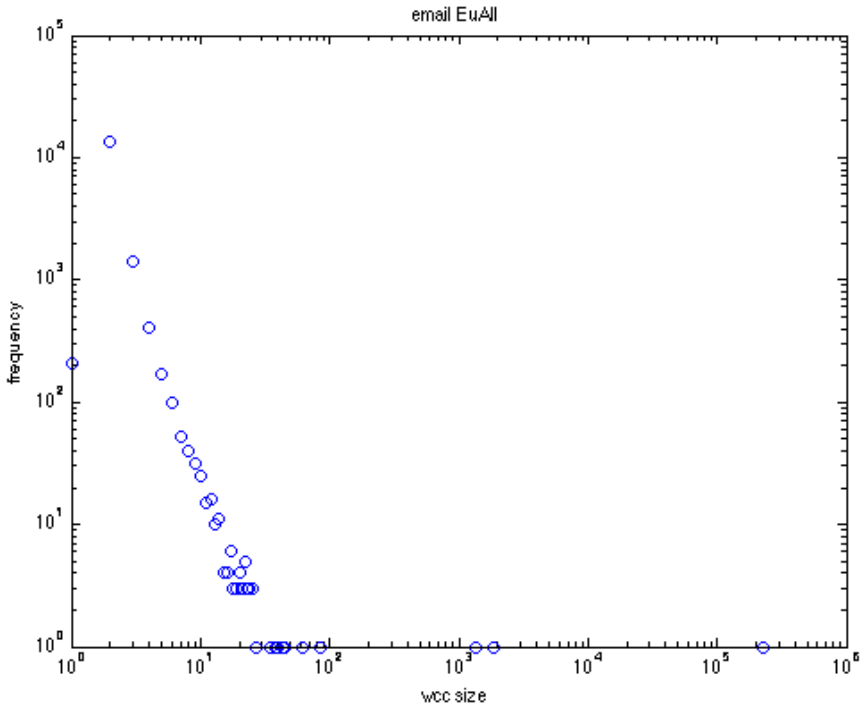



Figure 16: Email-EuAll

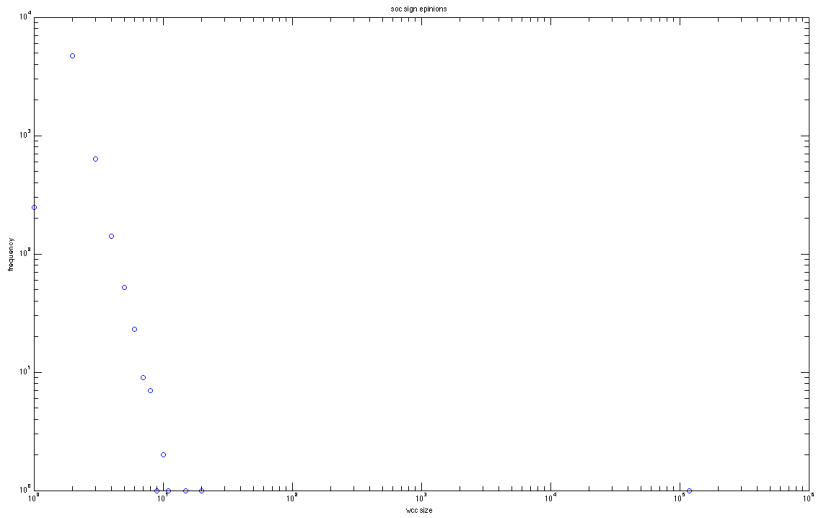



Figure 17: Soc-Sign-Epinions

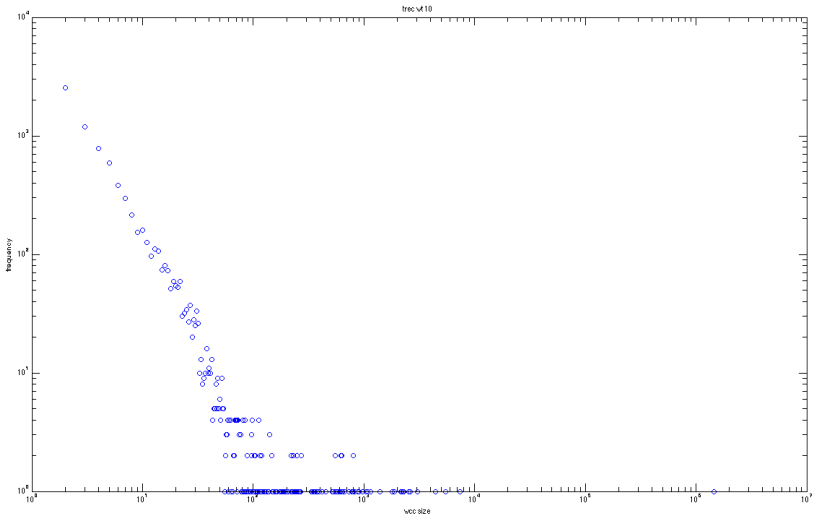


Figure 18: Trec-wt10g

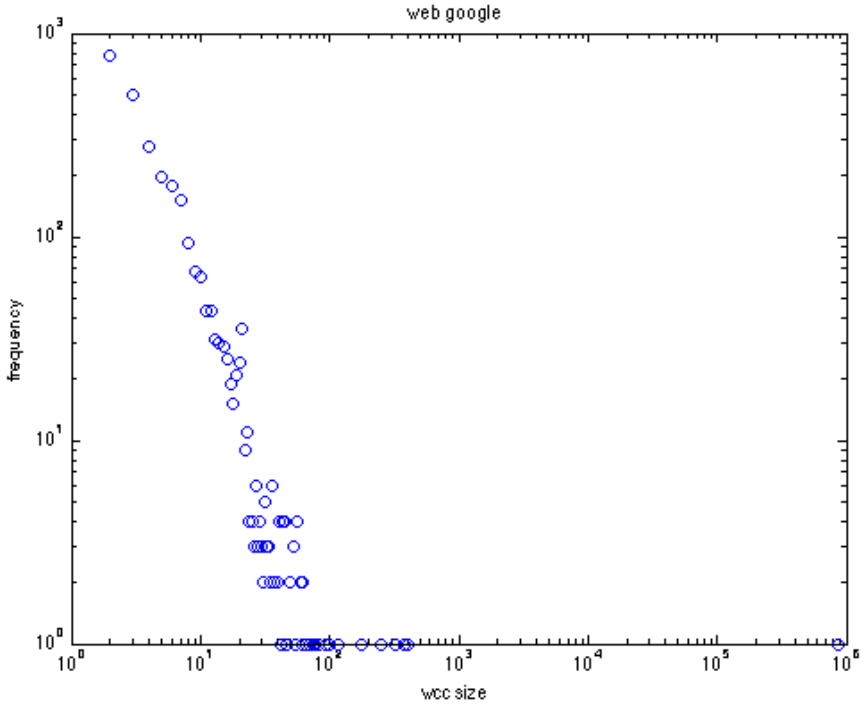



Figure 19: Google Web Graph

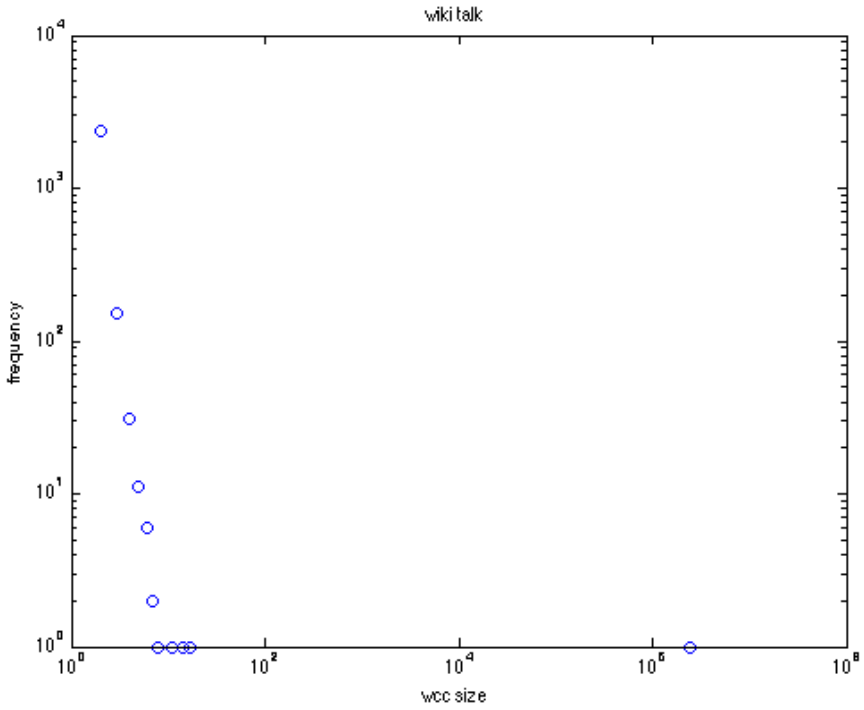



Figure 20: Wiki Talk

Table 3: Weakly Connected Component Run Result

| graph | number of nodes | number of edges | number of wcc | nodes in largest wcc |
|---|---|---|---|---|
| com-Youtube | 1134890 | 2987624 | 1 | 1134890 |
| web-Google | 875713 | 5105039 | 2746 | 855802 |
| com-Amazon | 334863 | 2987624 | 1 | 1134890 |
| email-EuAll | 265214 | 420045 | 15836 | 224832 |
| wiki-talk | 2394385 | 5021410 | 2555 | 2388953 |
| soc-sign-epinions | 131828 | 841372 | 5816 | 119130 |
| trec wt10g | 1601787 | 8063026 | 7955 | 1458316 |

### 4.4.3 Observation

1. From the log-log scale frequency-size plot, we find that generally frequency-size follows the power law. Connected components of small sizes tend to occur more often those of larger sizes.

2. In all the plots, there is a giant connected component that contains the majority of

nodes in the graph. This is a characteristic that almost all large graphs share, for example, in the yahoo web graph, there is also a giant connected component.

## 4.5 Task 4: Radius of Every Node

### 4.5.1 Experiment on large datasets

In this experiment, we run our radius algorithm in several large datasets. The statistics of these datasets are presented in Table 4

Table 4: Datasets Statistics

| dataset | number of vertices | number of edges | diameter |
|---|---|---|---|
| DBLP co-authorship network | 317080 | 1049866 | 21 |
| Epinions social network | 131828 | 841372 | 14 |
| Amazon product co-purchasing network | 334863 | 925872 | 44 |
| EU email communication network | 265214 | 420045 | 14 |
| Google web graph | 875713 | 5105039 | 21 |
| Youtube social network | 1134890 | 2987624 | 20 |

### 4.5.2 Observation

1. Radius Distribution: From the above plot, we find radius distribution of these graphs either tend to be single-modal, for example, EU Email Communication graph in figure 21 and Epinions Social Network in figure 22, or bi-modal like DBLP co-authorship network in figure 23 and Youtube Social Network in figure 25.

2. Relationship to the connected component: So why radius tend to be distributed like this, we guess it's somehow related to the connectivity of the graph. Therefore we take a look back to the statistics we got from task3. We find that the graphs that has a single-modal shape radius distribution are fully or almost fully connected. The graphs that has bi-modal radius distribution are not that well connected, For most nodes that appear in the Giant Connected Component(GCC), they tend to have a higher radius value, specifically the smaller radius value it has, the more centric it is in the GCC. On the other hand, The first peak in the radius distribution represent those disconnected components.

### 4.5.3 Proof of Correctness

Since the algorithm we use is an approximate algorithm, it's hard for us to verify the validity of our algorithm accurately. But we still try the algorithm on both small, large and synthetic datasets to demonstrate its validity. First we test the algorithm on the synthetic tiny dataset consisting of only 10 nodes and 8 edges. It accurately compute the radius of every node.

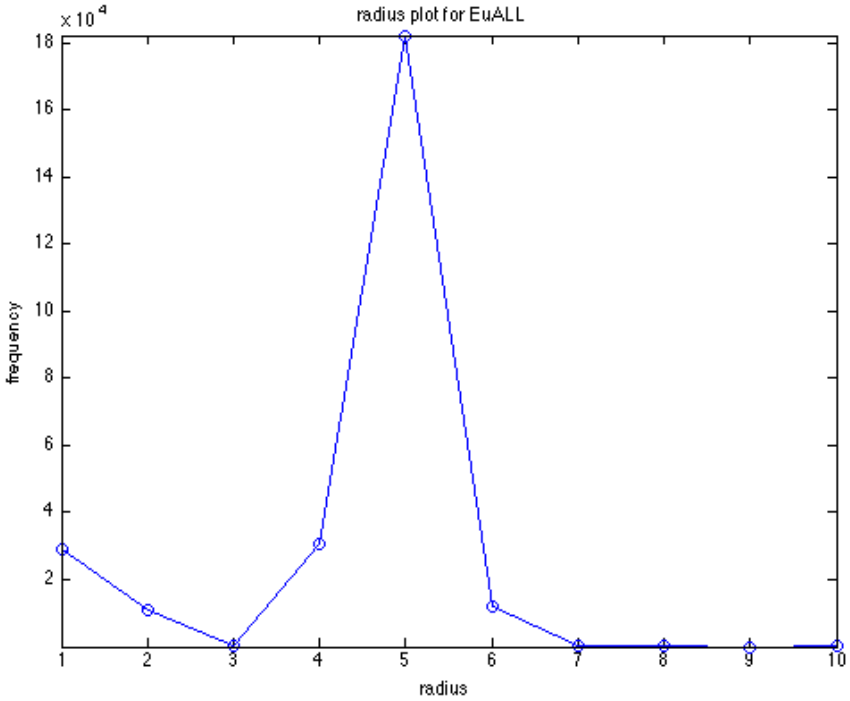



Figure 21: EU Email Communication

Then we test our algorithm on small datasets of several thousands of nodes, the experiment result are showed in Table 5, we can see that our algorithm can get a closer estimate of diameter.

Table 5: Radius Experiment on Small Datasets

| dataset | nodes | edges | real diameter | estimated diameter |
|---|---|---|---|---|
| Enron email network | 36692 | 183831 | 11 | 11 |
| High Energy Physics | 12008 | 118521 | 13 | 12 |
| Advogato Trust Network | 6551 | 51332 | 9 | 9 |

For large dataset we run in last section, the estimated diameter is still bounded by real diameter, but the estimation is no longer that close. This can be explained by the approximation nature of the algorithm we use. The Flajolet-Martin string we generate for every node is not unique, and as the number of nodes grows larger, the uniqueness of a node that a Flajolet-Martin string can represent is weakened. Therefore, for graph with larger size, the estimation might become less accurate.

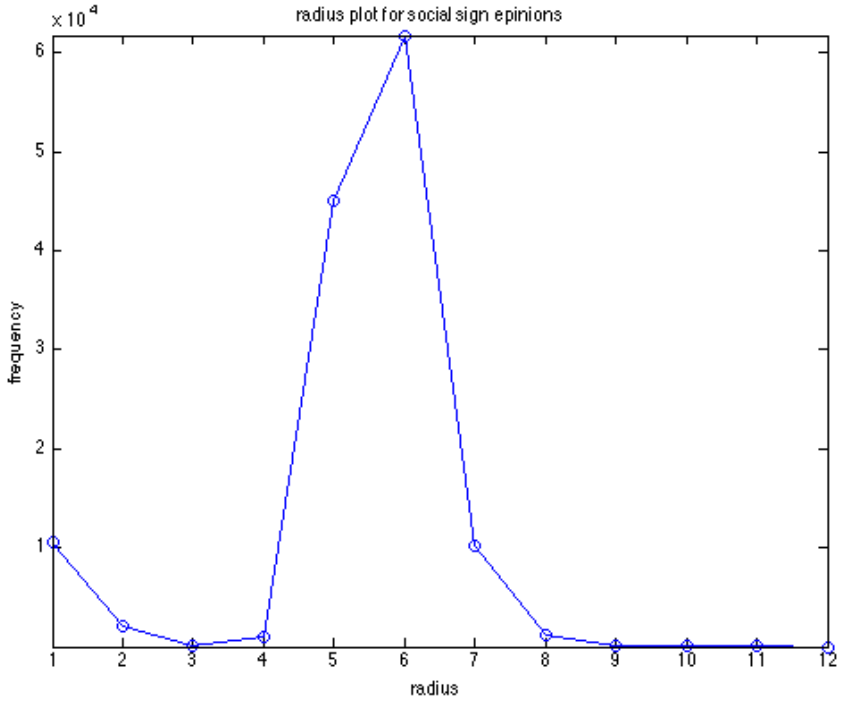



Figure 22: Epinions social network

## 4.6 Task 5: Eigenvalue/Singular value

In this experiment, we compute the eigenvalue of several datasets. The detailed description is in Table 6.

| Name | Nodes | Edges |
|---|---|---|
| wiki-Vote | 7,115 | 103,689 |
| youtube | 1,134,890 | 2,987,624 |
| slashdot0922 | 82,168 | 948,464 |
| com-DBLP | 317,080 | 1,049,866 |
| wiki-Talk | 2,394,385 | 5,021,410 |

Table 6: Dataset for Task 5

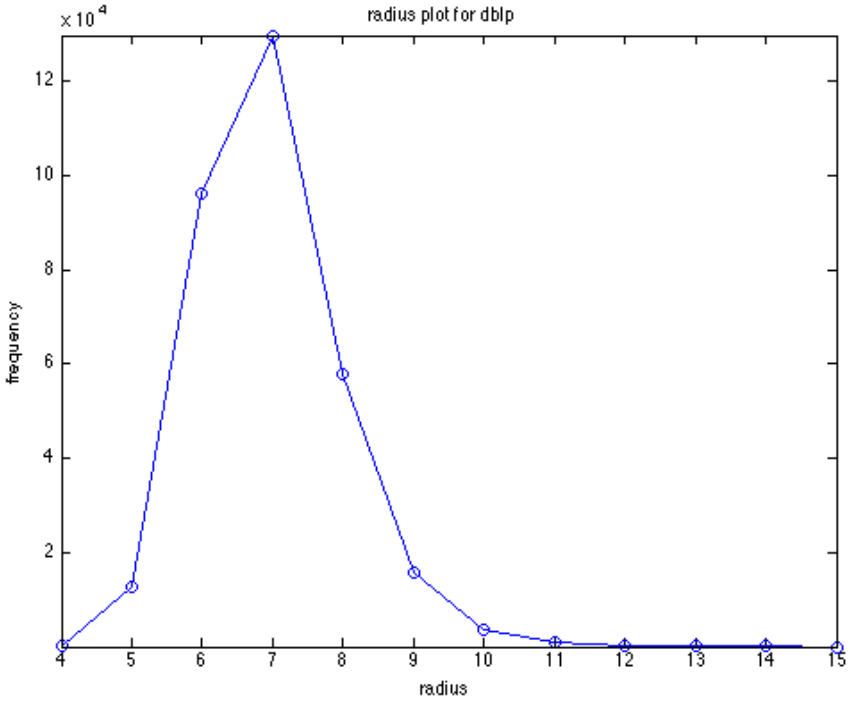



Figure 23: DBLP co-authorship network

### 4.6.1 Details

**Proof of Correctness:** In order to prove the correctness, we run our algorithm on the relatively smaller dataset, which is *Zachary karate club network*[3]. It has 34 nodes and 78 edges. The true top eigenvalues of Zachary is shown in Figure 27. We also calculated top-10 eigenvalues by our SQL implemention, the result is in Table 7. We can see that most of the top eigenvalues are near to its counter part in ground truth. So we are confident that our implementaiton is correct.

**wiki-Vote** The top 5 eigenvalues of wiki-Vote are 153.77, 107.23, 72.34, 49.23, -5.64. The plot(Figure 28) is as follows:

**youtube** The top 5 eigenvalues of Youtube are 238.55, 155.78, 83.29, 41.60, -7.5. The plot(Figure 29) is as follows:

**slashdot** The top 5 eigenvalues of slashdot are 125.34, 109.17, 85.23, 28.35, 4.34. The plot(Figure 30) is as follows:

[3]http://konect.uni-koblenz.de/networks/ucidata-zachary

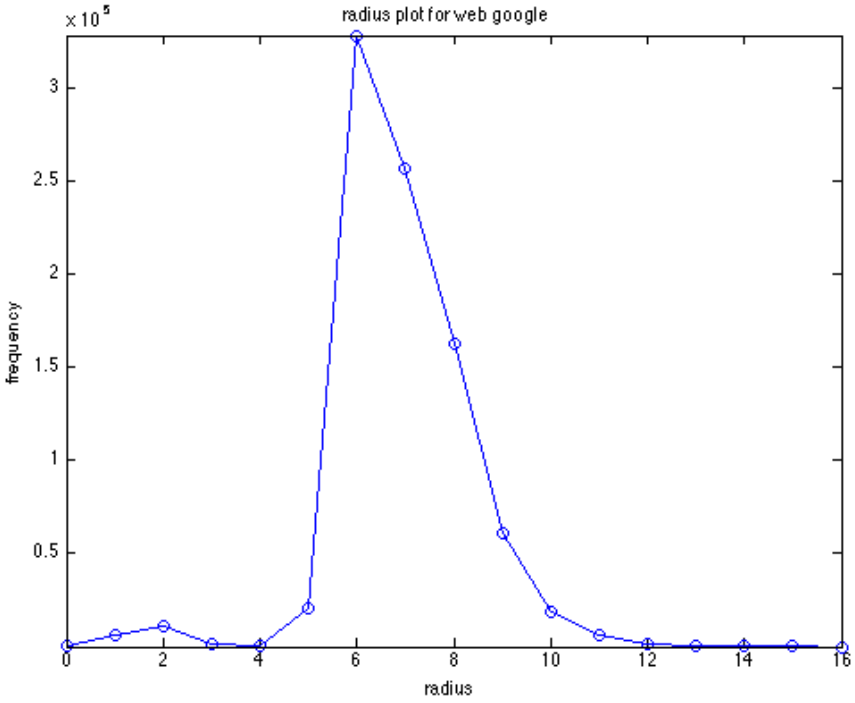



Figure 24: Google Web Graph

**com-DBLP** The top 5 eigenvalues of DBLP are 235.56, 173.45, 130.23, 95.14, 44.23. The plot(Figure 31) is as follows:

**wiki-Talk** The top 5 eigenvalues of Wiki-Talk are 353.23, 254.67, 125.82, 35.29, -33.45. The plot(Figure 32) is as follows:

### 4.6.2 Observation

We can see that in most datasets, the top eigenvalues drops quickly. This is also a reason why in Task 7, we only need top eigenvalues to count the number of triangles in a graph. Since the top eigenvalues is enough for the majority of the energy of the graph.

## 4.7 Task 6: Belief Propagation

### 4.7.1 Experiment on large datasets

In this experiment, we run our radius algorithm in several large datasets. The statistics of these datasets are presented in Table 8

Similar to semi-supervised learning, belief propagation algorithm require the graph to be partially labeled. However, we don't have such information. Therefore, in this experiment,

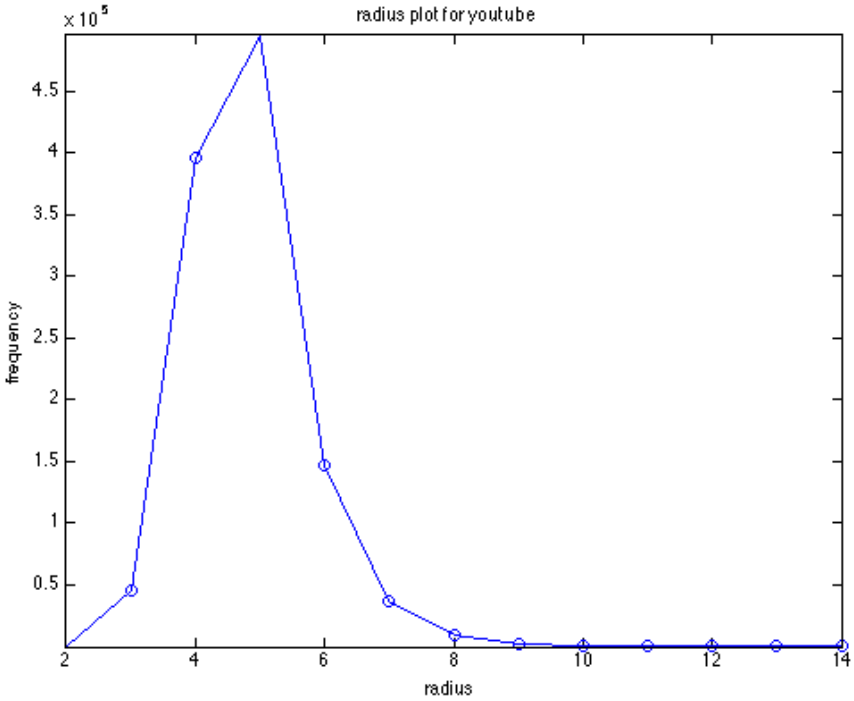



Figure 25: Youtube Social Network

we randomly assign the prior belief for all nodes. Specifically, we randomly assign 5% of the nodes with positive label, i.e. positive prior belief(0.001), and 5% of other nodes with negative label, i.e. negative prior belief(-0.001), and the rest with zero belief, meaning that we don't have prior knowledge for these nodes. Then we conduct FABP algorithm on these datasets, the result are shown in Table 9 .

### 4.7.2 Observation

1. By applying Belief Propagation algorithm on these graphs, most of the unlabeled nodes are successfully assigned either positive or negative belief.

2. We find that for some graphs, larger proportions of nodes gets labeled than other graph. For example, in DBLP co-authorship network and amazon product co-purchasing network, about 95% of the nodes get labeled using only 10% labeled nodes. While for like Epinions social network , only about 30% of the nodes get labeled. Once again, we look back at the connectivity of the graph to find the probable cause. We observed that, graph that is well connected is easier to get more nodes assigned with labels. This observation makes sense in that 'beliefs' can be easier to propagate in well connected graphs than those grape with many disconnected components.

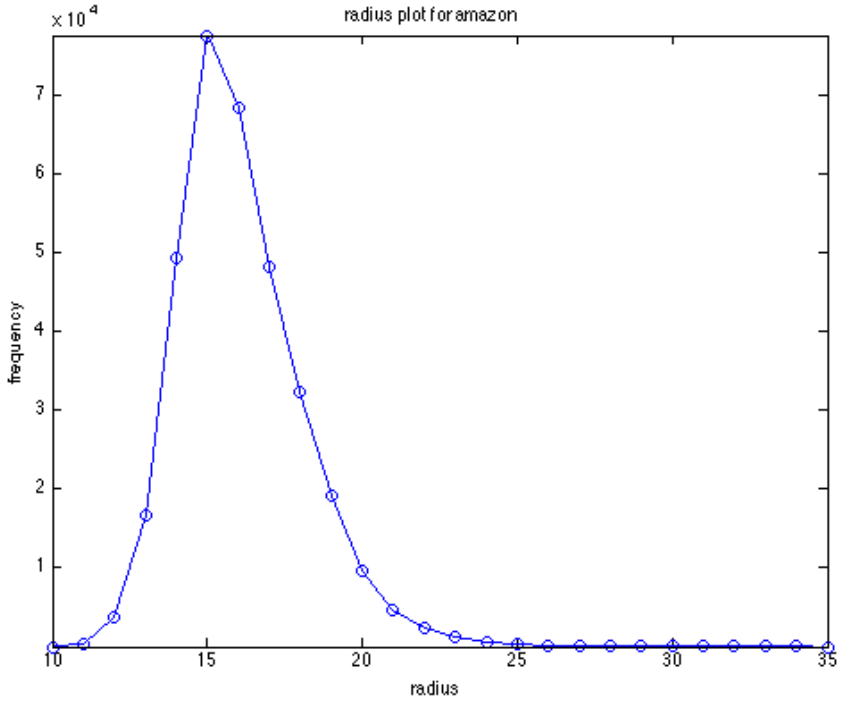



Figure 26: Amazon product co-purchasing networkk

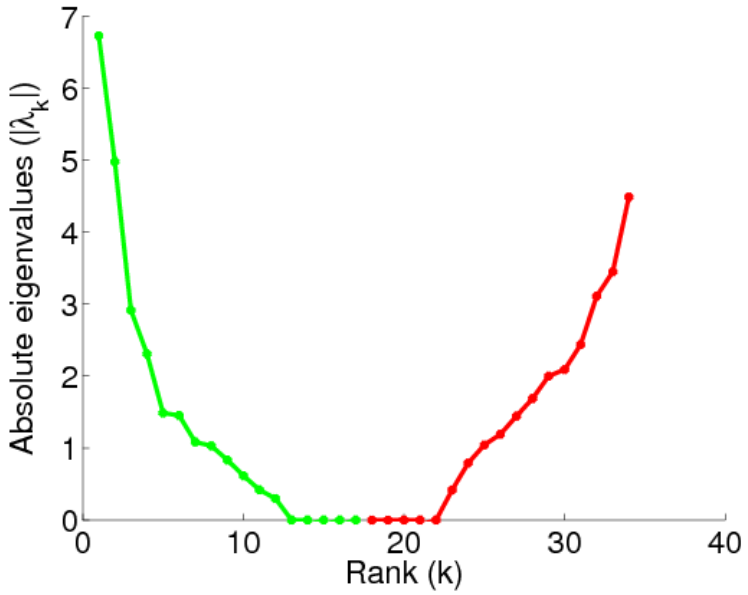



Figure 27: Top eigenvalues of Zachary

| Rank | Eigenvalue |
|---|---|
| 1 | 6.72569758513981 |
| 2 | 5.04694048054693 |
| 3 | 4.97613169036548 |
| 4 | 2.65295803734598 |
| 5 | 2.37853063223689 |
| 6 | 1.31746846315176 |
| 7 | 1.05904538243756 |
| 8 | 0.827084251502258 |
| 9 | 0.475203434096779 |
| 10 | 0.0182376036753603 |

Table 7: Top eigenvalues of Zachary

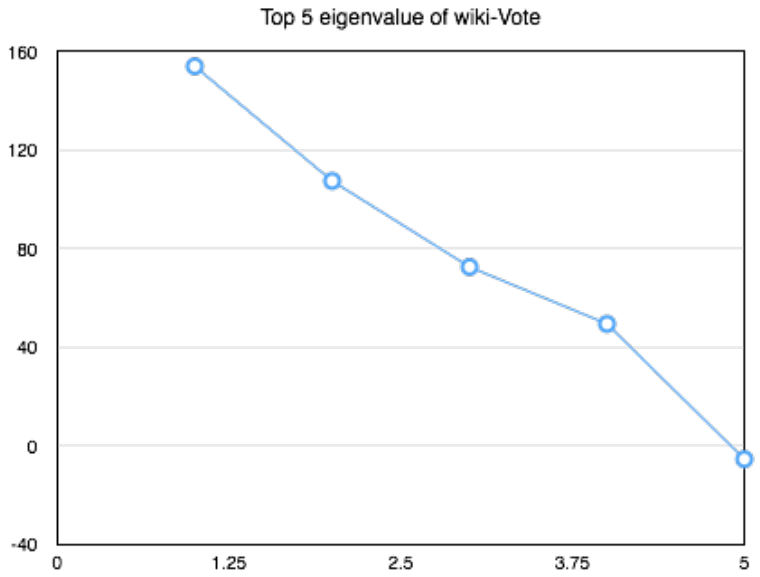



Figure 28: Top eigenvalues of wiki-Vote

### 4.7.3 Proof of Correctness

As mentioned in the previous section, we don't have any labels for these large datasets, therefore we verify the result according to statistics we got in the last section. We can see that the proportion with positive labels and negative labels are approximately same, which resembles the label distribution with our prior belief. Also, we successfully inferenced the belief of other nodes using only 10% labeled nodes. In order to verify the accuracy of our algorithm, we run FABP on small matrix we generate. In essence, the FABP tries to solve a linear system $(I - W)x = prior$, therefore we test our FABP against the linear system solver in MATLAB(function linsolve). And the two get nearly identical result(with error less than

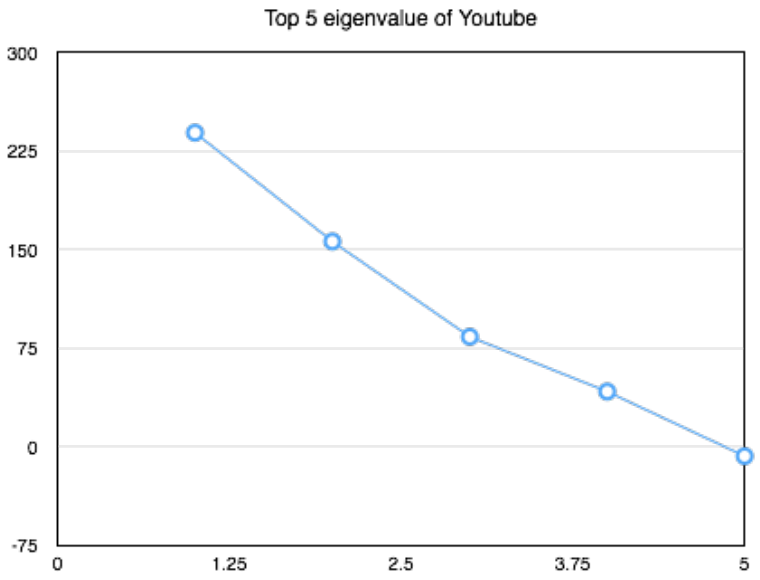



Figure 29: Top eigenvalues of Youtube

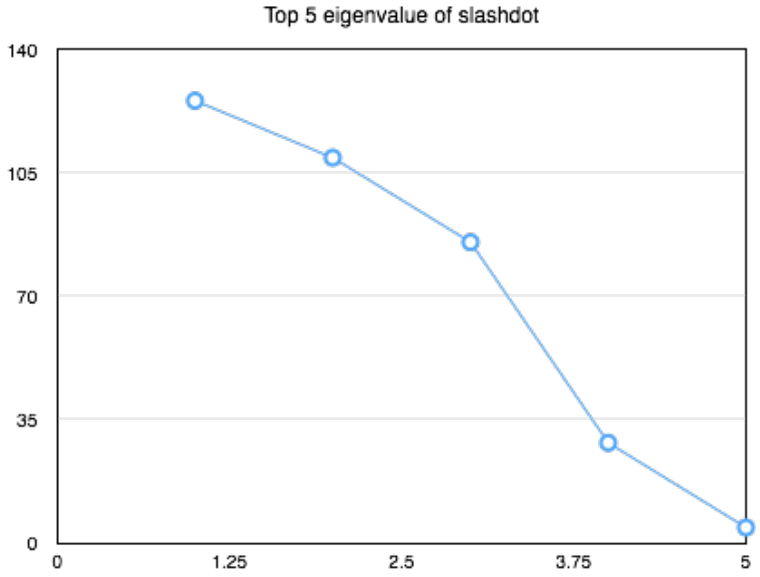



Figure 30: Top eigenvalues of Slashdot

0.01) in solving some toy linear systems consisting of 2 by 2 matrix. This demonstrates the correctness of our algorithm.

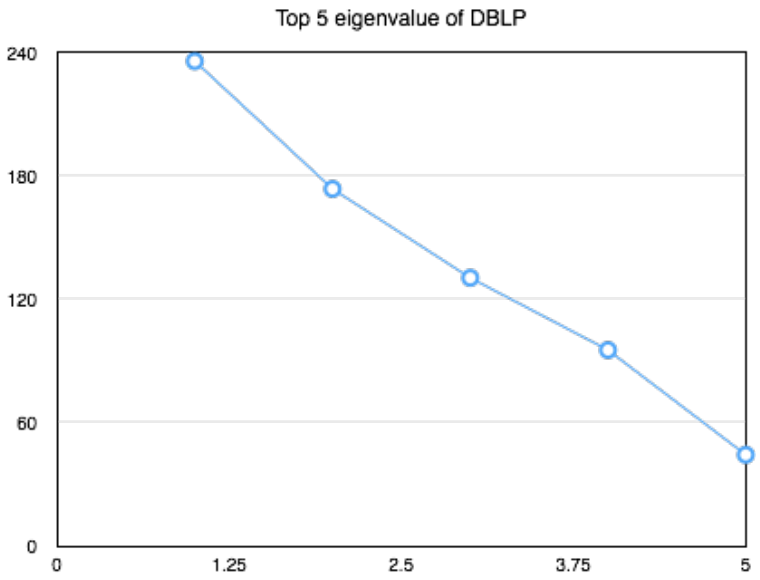



Figure 31: Top eigenvalues of DBLP

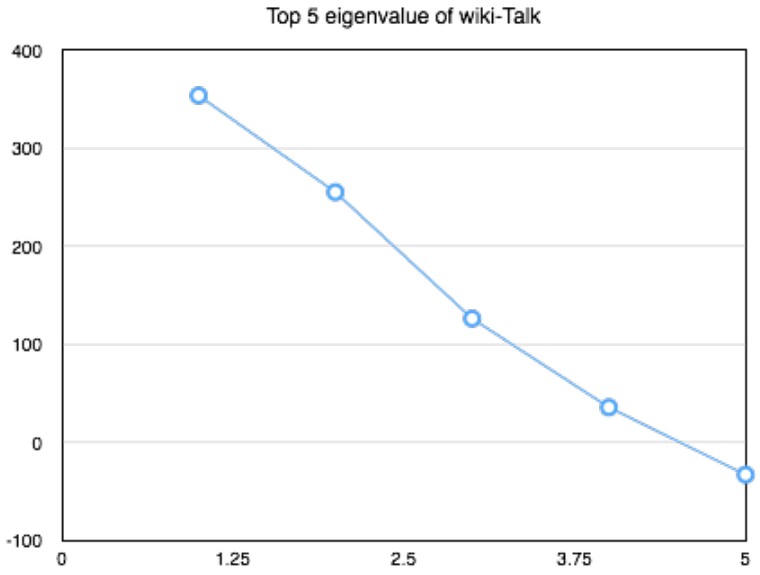



Figure 32: Top eigenvalues of wiki-Talk

## 4.8 Task 7: Triangle Counting

According to the algorithms in [14], we know that the number of triangles in a network is proportional to the sum of eigenvalue of its adjacency matrix, which is $\frac{\sum_i \lambda_i^3}{6}$. Figure 10 shows the running time of global triangle counting with regards to the size of graph. We can see that as the size of graph increases, the running time also grows nearly linearly with the

Table 8: Datasets Statistics

| dataset | number of vertices | number of edges |
|---|---|---|
| DBLP co-authorship network | 317080 | 1049866 |
| Epinions social network | 131828 | 841372 |
| Amazon product co-purchasing network | 334863 | 925872 |
| EU email communication network | 265214 | 420045 |
| Google web graph | 875713 | 5105039 |
| Youtube social network | 1134890 | 2987624 |

Table 9: BP Statistics

| dataset | positively labeled | negatively labeled | unlabeled |
|---|---|---|---|
| DBLP co-authorship network | 164103 | 144259 | 8718 |
| Epinions social network | 22334 | 22575 | 86919 |
| Amazon product co-purchasing network | 158837 | 160524 | 15502 |
| EU email communication network | 127481 | 109616 | 28117 |
| Google web graph | 385277 | 389975 | 100461 |
| Youtube social network | 584422 | 508628 | 41840 |

size. For the largest graph, which is Roadnet-PA, it runs nearly for an hour to complete. However, the predicted result for Roadnet-PA is unsatisfactory. We conduct both global and local triangle counting, all the data is listed as follows.

### 4.8.1 Detailed Plots

**Proof of Correctness:** In this experiment, we run our algorithm on the dataset, and verify that the predicted number of triangles is a good approximate of the true count. The full result is in Table 11. As we can see, most of the result is near to each other. So we are aure about the correctness of the implementation.

Table 10 lists run time of global triangle counting. Figure 33 plots the run time of global triangle counting on each dataset.

### 4.8.2 Local triangle counting

We plot the rank-frequency plot of local triangle counting, that x-axis represent the rank of the count of local triangle, y-axis represents the number of local triangles at that rank.

| graph size | run time(seconds) |
|---|---|
| 7115 | 45.199s |
| 36692 | 72.76 |
| 82168 | 596.046 |
| 334863 | 1288.703 |
| 1088092 | 2980.985 |

Table 10: Task 7 run time(global)

| dataset | size | predict | truth |
|---|---|---|---|
| wiki-Vite | 7115 | 661282 | 608389 |
| Enron-email | 36692 | 756757 | 727044 |
| slash-dot | 82168 | 635792 | 602592 |
| Amazon-com | 334863 | 675778 | 667129 |
| Roadnet-PA | 1088092 | 72134 | 67150 |

Table 11: Predicted triangle count(global)

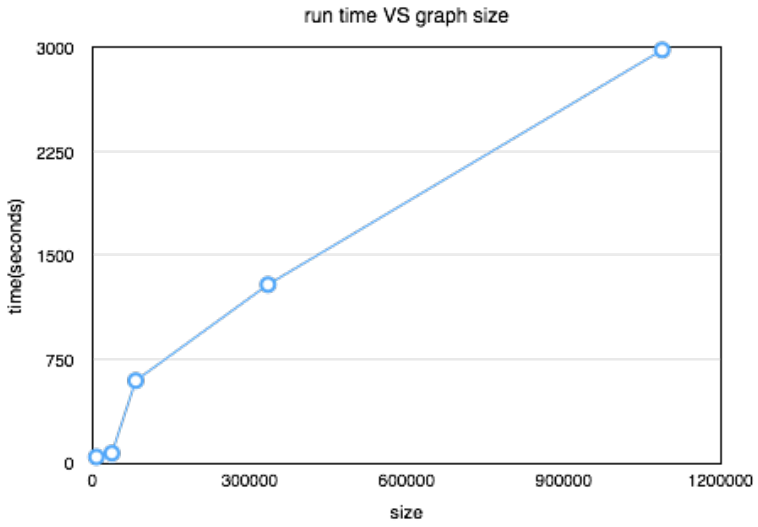



Figure 33: Task 7: Run time VS graph size(global)

**Amazon** Figure 34 plots the rank-frequency of Amazon. We can observe from the figure that it follows **power law**.

**Enron mail** Figure 35 plots the rank-frequency of Enron Mail. We can observe from the figure that it follows **power law**.

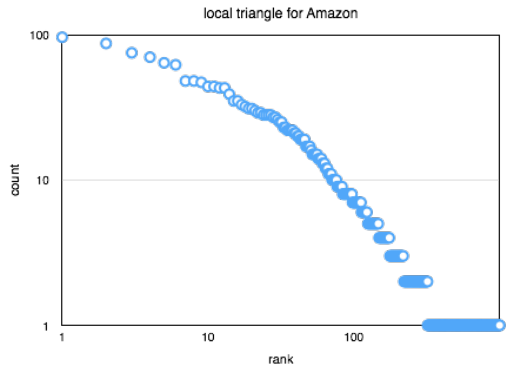



Figure 34: Local triangle counting for Amazon

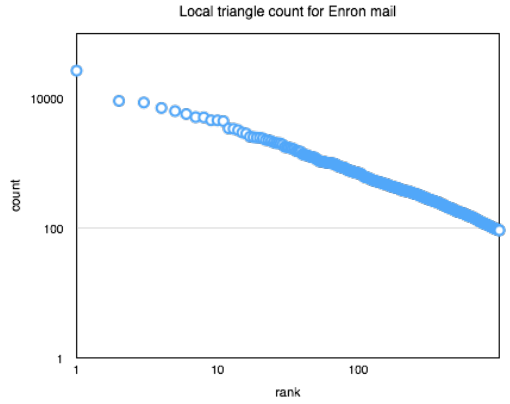



Figure 35: Local triangle counting for Enron mail

**Slashdot** Figure 36 plots the rank-frequency of Slashdot. We can observe from the figure that it follows **power law**.

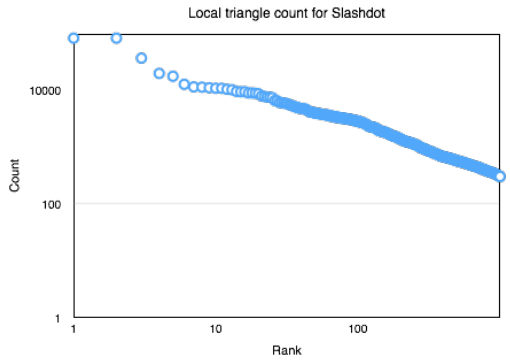



Figure 36: Local triangle counting for Slashdot

**wiki-Vote** Figure 37 plots the rank-frequency of wiki-Vote. We can observe from the figure that it follows **power law**.

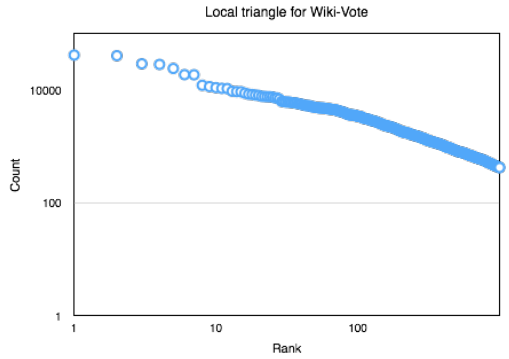



Figure 37: Local triangle counting for wiki-Vote

**Youtube** Figure 38 plots the rank-frequency of Youtube. We can observe from the figure that it follows **power law**.

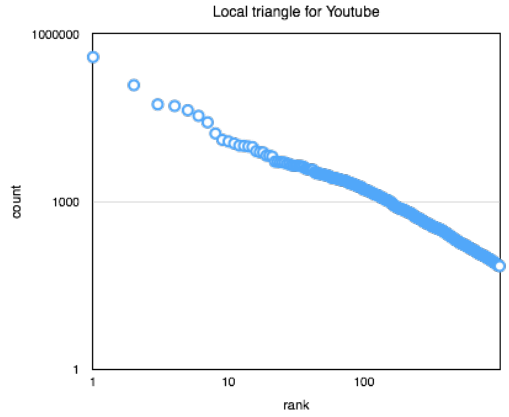



Figure 38: Local triangle counting for youtube

### 4.8.3 Observation

We can see that the rank-frequency plot of local triangle count also follows **power law**, it just matches our intuition. And we can observe that the run time grows nearly linearly with the graph size.

## 4.9 Innovative task: Shortest Path

**Proof of Correctness:** In order to prove the correctness, we conduct experiments on a small dataset that we manually constucted. And the result is exactly the same.

In order to test the performance of the SQL implementation, we run the shortest path algorithm on incrementally larger datasets, the running time erquired according the size of graph is ploted in Figure 39. We can observe from that figure that as the graph size increases, the running time grows much faster than linear. The reason for this is that we implemented all the data structures through tables, which incurs a lot of overhead of search. An alternative is to use the data types provided by Postgres.

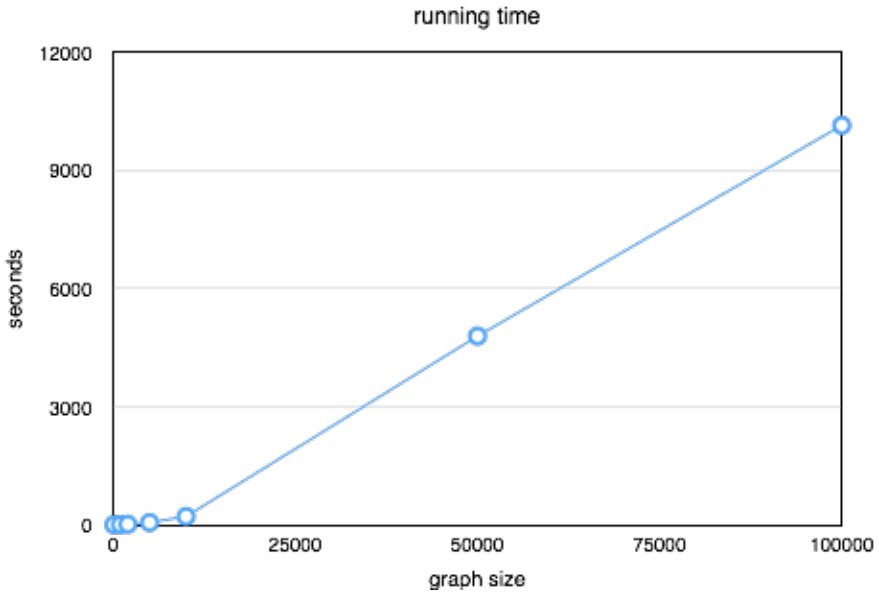



Figure 39: Running time VS graph size

## 4.10 Innovative task: Minimum Spanning Tree

### 4.10.1 Proof of Correctness

In order to verify the correctness of our implementation, we first run our algorithm on tiny synthetic graph with 10 nodes and 17 edges. We verified the result is correct. Then we compare our implementation against MATLAB's implementation of Minimum Spanning Tree and we get identical result for output of both implementation.

### 4.10.2 Experiment on Large datasets

Since MST algorithm require that graph is fully connected, weighted, and undirected. We can barely find such graph in Konect and SNAP project. Therefore, we generate synthetic

graph of different size. We plot the runtime against graph size (number of nodes) in Figure 40. We find the run time grows near-quadratically as the number of nodes in graph. This is identical to the time complexity of Prim's Algorithm, which is $O(N^2)$.

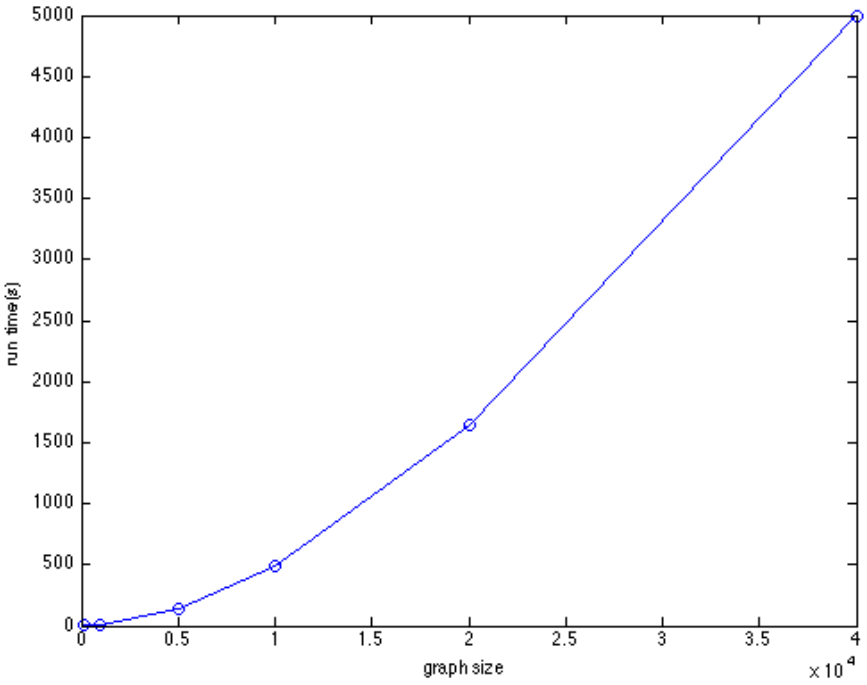



Figure 40: MST runtime plot

# 5 Conclusions

In this project, we investigate the major questions discussed in graph mining. We explored and solved the following problems:

- We summarize the importance of graph mining techniques and propose our approach to this problem, which is Relational Database Management System.
- We conduct extensive survey about 7 graph mining tasks, each team member has read at least six papers each for the tasks.
- For task 1, we calculate degree distribution of each node and do visualization. By observing the plot, we conclude that **social network** data exhibits **power law**, while this is not a general rule, for instance, Roadnet dataset doest not show power law.

- For task 2, we calculate the pagerank of each node in a graph, namely the importance of each node. By visualize the pagerank distribution in a rank-frequency plot, we again observe **power law**.
- For task 3, we compute the weakly connected components for a graph. By observing the plot, we find that there is a Giant Connected Component(GCC) which contains the majority of nodes in a graph. And the frequency-size plot exhibits **power law**.
- For task 4, we calculate the radius for every node in a graph using an approximate algorithm. By visualizing the result as radius plot, we find that most graphs has a **single-modal** or **bi-modal** radius distribution.
- For task 5. We tackle the problem of calculating eigenvalue of an adjacent matrix by Lanczos mathod. We build a Python matrix operation library which wraps low level linear algebra operations of SQL.
- For task 6. We implement the belief propagation using FABP algorithm. By conducting experiments on large datasets, we successfully perform **semi-supervised learning** using partially available labels to inference the labels of other nodes in the graph.
- For task 7. We deal with the problem of count triangle(global or local) in a graph by calculating the eigenvalue of the adjacent matrix. Through the rank-frequency plot of local triangle distribution, we again observe **power law**.
- We finished 2 innovative tasks, namely shortest path, minimum spanning tree.
- We provide proof of correctness for each task we implemented.

# References


[1] P.F. Felzenszwalb and D.P. Huttenlocher. Efficient belief propagation for early vision. *International journal of computer vision*, 70(1):41–54, 2006.

[2] U. Kang, Duen Horng Chau, and Christos Faloutsos. Mining large graphs: Algorithms, inference, and discoveries. In *ICDE*, pages 243–254, 2011.

[3] U Kang, Duen Horng, et al. Inference of beliefs on billion-scale graphs. 2010.

[4] U. Kang, Mary McGlohon, Leman Akoglu, and Christos Faloutsos. Patterns on the connected components of terabyte-scale graphs. In *ICDM*, pages 875–880, 2010.

[5] U Kang, Brendan Meeder, and Christos Faloutsos. Spectral analysis for billion-scale graphs: Discoveries and implementation. In *Advances in Knowledge Discovery and Data Mining*, pages 13–25. Springer, 2011.

[6] U Kang, Charalampos Tsourakakis, and Christos Faloutsos. PEGASUS: A peta-scale graph mining system - implementation and observations. *ICDM*, December 2009.

[7] U. Kang, Charalampos E. Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. Hadi: Mining radii of large graphs. *TKDD*, 5(2):8, 2011.

[8] Danai Koutra, Tai-You Ke, U. Kang, Duen Horng Chau, Hsing-Kuo Kenneth Pao, and Christos Faloutsos. Unifying guilt-by-association approaches: Theorems and fast algorithms. In *ECML/PKDD (2)*, pages 245–260, 2011.

[9] Cornelius Lanczos. *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office, 1950.

[10] Amy N Langville and Carl D Meyer. Deeper inside pagerank. *Internet Mathematics*, 1(3):335–380, 2004.

[11] Robert J. McEliece, David J. C. MacKay, and Jung-Fu Cheng. Turbo decoding as an instance of pearl's belief propagation algorithm. *Selected Areas in Communications, IEEE Journal on*, 16(2):140–152, 1998.

[12] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: a not-so-foreign language for data processing. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1099–1110. ACM, 2008.

[13] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J Abadi, David J DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 165–178. ACM, 2009.

[14] Charalampos E Tsourakakis. Fast counting of triangles in large real networks without counting: Algorithms and laws. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*, pages 608–617. IEEE, 2008.

[15] J.S. Yedidia, W.T. Freeman, and Y. Weiss. Understanding belief propagation and its generalizations. *Exploring artificial intelligence in the new millennium*, 8:236–239, 2003.

# A Appendix

## A.1 Labor Division

The team performed the following tasks

- Implementation of Task 1,2 [Wei Chen]
- Implementation of Task 3 [Siping Ji]
- Implementation of Task 4,6. [Siping Ji]
- Implementation of Task 5,7. [Wei Chen]
- Implementation of Shortest Path(Innovative). [Wei Chen]
- Implementation of Minimum Spanning Tree. [Siping Ji]
- Set up testing framework in travis[4]. [Wei Chen]
- Data collection [Wei Chen, Siping Ji]
- Experiments on the real data [Wei Chen, Siping Ji]

## A.2 Project Development

All the project related activity(code, report) are managed by Github[5], a collaborative development community. All tasks are mapped into *issues*, each phase is a *milestone*. We fork from a central repository, when we finish our assigned tasks(issue), we send a pull request to the central repository. We are responsible for code review each other's code, then merge into the central repository. It's easy to see each team member's contribution by review the history of pull request and commit. The project address is here.[6]

## A.3 Plan

Phase 1

| Task | Due | Member |
|---|---|---|
| Task1 | 10/07/13 | Wei |
| Task2 | 10/07/13 | Wei |
| Task3 | 10/07/13 | Siping |

Phase 2

| | | |
|---|---|---|
| Task4 | 11/05/13 | Siping |
| Task5 | 11/05/13 | Wei |
| Task6 | 11/05/13 | Siping |
| Task7 | 11/10/13 | Wei |
| Task8 | 11/10/13 | Siping |
| Final | 11/20/13 | Wei, Siping |
| Poster | 11/20/13 | Wei, Siping |

[4]travis-ci.org

[5]https://www.github.com

[6]https://github.com/essex405/graph-mining-rdbms

Phase 3

| | | |
|---|---|---|
| Packaging code | 11/30/13 | Wei Chen, Siping |
| Final report | 11/30/13 | Wei Chen, Siping |

## A.4 Additional Task

In addition to the default tasks, we plan to complete another two: shortest path and minimum spanning tree. The detailed report please refer to **Experiments**.

## A.5 Unit Tests

The host language we use is Python, thus we plan to use its internal unit test framework[7] as our testing module. We did unit tests for following modules:

- Matrix Vector multiplication, abstract out the some basic matrix-vector, matrix-matrix operations.
- SQL user defined function bit-or
- SQL user defined function fm-size

[7] http://docs.python.org/2/library/unittest.html

# Contents