Graph Mining using SQL

Yuning He

Dept. of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
yuningh@andrew.cmu.edu

Kuo Liu

Dept. of Computer Science
Carnegie Mellon University
Pittsburgh, PA, USA
kuol@andrew.cmu.edu

Abstract. Graph mining is an important branch of Data Mining. It is mainly focused on mining some useful information from graphs. There are some basic tasks in graph mining which is a good start for beginners. In this project, we make use of some algorithms proposed by several papers in this area and develop a system for solving these basic problems in graph mining. We mainly use RDBMS to achieve our tasks, which is a promising research direction.

Keywords: Graph mining. RDBMS.

1 Introduction

There are lots of networks in our society such as transportation network, social network. Nowadays, a new kind of network, online social network, emerges due to the booming Internet. These networks contain lots of information which we can make use of to improve our life quality. Since various kinds of networks can be abstracted as graphs, graph mining is a very important area to do research on. There are some basic problems (although basic in concept, it is not always simple) in graph mining which is a good start to step into this area. The problems we want to solve are as follows:

- GIVEN: a file specifying the edge list of a graph.
- FIND:

  1. Degree Distribution of the nodes.
  2. PageRank score of the nodes.
  3. The proximity of nodes in graph using Random Walk with Restart. (Extra task for Kuo Liu)
  4. Connected Components (weak) in the graph.
  5. Radius, for every node. Use the Martin-Flajolet method in 'HADI'.
  6. Eigenvalues/singular values: Use the method in HEIGEN.
  7. Belief Propagation: Use the binary, "Fast" belief propagation.
  8. Count of Triangles: Use the wedge sampling approximation algorithm.
  9. Shortest Path: Dijkstra algorithm (Extra task for Yuning He).
  10. Broad-spectrum graph mining.

- To MAXIMIZE: the number of tasks above implemented, the datasets from SNAP/KONECT/Newman applied on our system.

Even though these problems may seem to be simple at the first glance, things get much more difficult when the graphs become very large, especially that the graphs in some fields like social network have become billion-scale. The time efficiency and space efficiency of the algorithm become very important in this situation.

The contributions of this project are the following:

- We solved the basic tasks using algorithms proposed by some papers published on some top conferences and journals based on top of RDBMS.
- We extend some algorithms so that they can be applied on directed graphs.
- We have done experiments on several datasets and observed some interesting phenomena.

2 Survey

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

2.1 Papers read by Kuo Liu

The first three papers are all by U. Kang [1][2][3].
The first paper is mainly focused on mining peta-scale graphs [1].

- Main idea: This paper mainly focuses on some basic graph mining tasks such as computing the page rank score of nodes, finding connected components and so on. There are two novel points presented in this paper.
  The first is that Kang unified these tasks with GIM-V (Generalized Iterative Matrix-Vector multiplication), which is a different perspective to see these graph mining tasks. He showed us that all these tasks can be divided into three basic operations, which are combine2, combineAll and assign.
  Another point is that he applied the idea above in the Map-Reduce framework, made use of Hadoop to perform these tasks on some peta-scale graphs and did some optimizations. We see good speedup and scalability of this method in the experiments.
- Use for our project: This paper is extremely related to our task since we also have to compute the page rank score of each node and find connected components using GIM-V. This paper explains how we can use SQL to achieve these tasks.
- Shortcomings: Kang didn't propose how to parallel in RDBMS when applying GIM-V.

The second paper is mainly focused on mining radii of large graphs [2].

- Main idea: This paper focuses mainly on computing the radius of each node in a large graph, which is also a basic task in graph mining. Since computing the exact radius for each node in a large graph is impractical, Kang made use of Flajolet-Martin algorithm to do this kind of job. Flajolet-Martin algorithm is for estimating

the number of unique elements in a multi-set based on the characteristics of good hash function. Kang made use of this method to estimate the number of neighbors for each node.
  Another point in this paper is to apply this algorithm in the framework of Map-Reduce and do some optimizations on HADI-Naive. The main optimization is to replicate bitstrings of node j exactly x times where x is the in-degree of node j. In this way, Kang reduced the space complexity. We see good speedup and scalability for this method in the experiments.
- Use for our project: We can also make use of Flajolet-Martin algorithm to compute the radius of each node. Moreover, this paper has a discussion on how HADI works in Parallel DBMSs. So, we can implement the methods proposed in this paper using SQL.
- Shortcomings: Before applying this algorithm, we should carefully choose a good hash function because the estimation might be damaged if we choose a hash function that is not suitable.

The third paper is mainly focused on computing eigenvalues and eigenvectors of billion-scale graphs [3].

- Main idea: In this paper, Kang made use of Lanczos-SO as the sequential eigen-solver algorithm to compute top k eigenvalues. The main idea of Lanczos-SO is to transform the original matrix into a tridiagonal matrix to get good approximations of the eigenvalues of the original matrix.
  Another point proposed in this paper is to do selective parallelization, just like the other two papers, the new algorithm also makes use of Hadoop to implement the parallelization.
- Use for our project: We can use the algorithm proposed in this paper to compute the eigenvalues of a large graph.
- Shortcomings: It does not cover the content about how to implement this method using RDBMS.

2.2 Papers read by Yuning He

The fourth paper is: *"GBASE : an efficient analysis platform for large graphs"* [4]

- Main idea: In this paper, the authors focus on building an efficient platform to solve the problems in large graphs, such as graph storage, operations, and query optimization.
  For storage, they propose a method to index and store the large graph on homogeneous blocks. First, they divide the graph by reordering rows and columns to form homogeneous blocks, which means that the sub-graphs can be dense or sparse. Then, they compress the blocks using a different encoding method. They convert the adjacent matrix of the graph into a binary string and store the compressed string so that they can save much space. Finally in the file systems, they store the compressed blocks in a grid placement fashion by which they can efficiently reduce the

number of disk accesses when handling both in-neighbors and out-neighbors queries.
For graph operations, the platform supports eleven types of graph queries using matrix-vector multiplications. Each query can be translated into a matrix-vector version and a corresponding SQL version using adjacency and incidence matrices, thus the query responses can be accelerated.
For query optimization, they build a query execution engine on top of Hadoop to achieve parallelism and fault tolerance. Based on the storage model and the matrix-vector multiplication fashion, they can use a grid selection algorithm and make use of adjacent matrix through Map-Reduce to support incidence matrix queries.

- Use for our project: In our project, we are also facing some very large data sets which may not fit in memory. So we may need some methods to interpret the original data into other structures. What's more, the matrix-vector manipulation for graph queries in this paper is also a possible implementation in order to achieve a faster response speed in such a big input data set.
- Shortcomings: Actually this method is built on top of Hadoop. So how to achieve the same acceleration and efficiency in RDBMS may be a problem.

The fifth paper is: *"Mining Large Graphs: Algorithms, Inference, and Discoveries"* [5]

- Main idea: In this paper, Kang intends to find patterns and anomalies in large graphs. He proposed a parallel method to do inference using intuitive "guilt by association" scenarios on top of Hadoop platform.
  Belief Propagation (BP) is an efficient algorithm to infer properties in probabilistic graph models. It iteratively updates the probability of the property in each node until the belief converges. But when facing large scale datasets, they need to change the BP algorithm a little bit to fit in the Map-Reduce framework and achieve tractable recursive equation. First, they change the original undirected graph to a directed graph. Then they derive the recursive equation from the original BP algorithm to the Line Graph Fixed Point (LFP) algorithm and the convergence condition is changed to find a fixed point. But the naive LFP algorithm will produce too many edges. So they propose a lazy multiplication method by which they do the computation in the original graph instead of the derived graph. Finally, the algorithm can fit into Hadoop platform and become HA-LFP algorithm.
- Use for our project: The LFP algorithm is powerful in recognizing patterns in graphs in parallel. In our project, we also have a task to do Belief Propagation on large datasets. So we are considering using LFP algorithm to reduce the computation work to achieve scalability.
- Shortcomings: Changing from BP to LFP needs some pre-processing. For large graphs, this step may also cost a lot of time. Moreover, LFP generates much more edges in line graph. How to apply lazy multiplication to RDBMS is what we should consider.

The sixth paper is: *"Counting Triangles in Real-World Networks using Projections"* [6].

- Main idea: In large scale graphs, the problem of estimating the clustering coefficients and the transitivity ratio of the graph can be translated into computing the triangles that each node participates or the total number of triangles in the graph. The solutions to this problem can be used for spam detections, community discoveries, or link recommendations. The authors propose two efficient algorithms to count the number of triangles, while they apply the algorithms to FastSVD in order to do computations on real-world, large networks that cannot fit in the main memory.
  The EIGENTRIANGLE and EIGENTRIANGLELOCAL algorithms are proved to be successful for real-world networks considering some special properties. Lanczos method is an underlying eigen-decomposition algorithm that can converge fast to the top k eigenvalues in a real-world networks, because they correspond to roots of the secular function that are well separated. The experimental result shows that these two algorithms are much faster for at least 95% accuracy. To deal with the problem that the graphs cannot fit into the main memory, the authors apply EIGENTRIANGLE to a FastSVD algorithm to count triangles.
- Use for our project: In our project, we also have a related task to count the number of triangles in a large real-world graph. We can also use the related techniques to estimate eigenvalues in other matrix-related operations.
- Shortcomings: The algorithms are implemented in Matlab, so how to efficiently use the algorithm in RDBMS is unresolved.

3 Proposed Method

We implemented nine fundamental graph mining algorithm using SQL and we use POSTGRESQL which is claimed to be the world's most advanced open source database as our RDBMS. These algorithms can all be found in our reading list except the extra task from Yuning and they represent the most commonly used methods used in graph mining in the industry world. The algorithms are listed as follows and we will discuss each one in detail in the following parts.

- Degree Distribution: About computing the degree (sum of in and out degree) of each node and plotting the degree distribution.
- PageRank: Computing the importance of each node in a graph using link analysis.
- Random Walk With Restart: Measure the proximity of nodes in graph.
- Connected Components: Partite one graph into groups while in each group each two nodes are connected via edges.
- Radius for Each Node: Finding the number of hops for each node to reach another farthest-away node.
- Eigenvalues/Singular Values: Computing the eigenvalues/singular values for the graph adjacency matrix.
- Belief Propagation: Classifying on probabilistic graphical models using the binary, efficient inference algorithm.

- Count of Triangles: Counting the number of triangles in an undirected graph using the (award winning) sampling approach.
- Shortest Path: Find single-source shortest paths using Dijkstra algorithm.

3.1 Basic Matrix Operations

Before explaining each algorithm in detail, we have to define some basic matrix/vector operations here. We see that many algorithms in graph mining can be defined in the format of some kinds of operation on matrixes and these general operations can be easily translated into SQL. Since much optimization has already been done on RDBMS, translating these basic operations into SQL and making use of RDBMS to do graph mining often help in many ways.

The first commonly used matrix operation is multiplication between matrix and vector, the algorithm is defined as follows.

Algorithm 1. Matrix-Vector Multiplication (**v′** = **M** * **v**)

```
Input: Matrix M, Vector v.
Output: Vector v'.
1: select M.row_id, sum(M.val * v.val) from M, v
2: where M.col_id = v.id
3: group by M.row_id;
```

One point to emphasize here is that this algorithm is just traditional matrix-vector multiplication. More general matrix-vector multiplication is more flexible and can be applied in various kinds of situations. Suppose we have a *n* by *n* matrix *M* and a vector ***v***. let $m_{i,j}$ denote the (*i*,*j*)-th element of *M*. Then we can divide the traditional matrix-vector multiplication into three parts.

1. *combine2*: multiply $m_{i,j}$ and v_j.
2. *combineAll*: sum *n* multiplication results for node *i*.
3. *assign*: overwrite the previous value of v_i with the new result to make v_i'.

General matrix-vector multiplication is summarized as follows.

Algorithm 2. Generalized Matrix-Vector Multiplication

```
Input: Edges E, Vector V.
Output: Vector V'.
1: select E.sid, $combineAll_{E.sid}$(combine2(E.val, V.val)) from E, V
2: where E.did = V.id
3: group by E.sid
```

We can see that several algorithms in the following parts follow the general format of this algorithm. The difference is just about the specific definition of the three basic operations.

Some other common operations are listed as follows:

Algorithm 3. Vector-Vector Addition (**y** = **y** + a**x**)

```
Input: Vector v1, Vector v2, a.
Output: Vector v1.
1: update v1 set v1.val = v1.val + a * v2.val from v2
2: where v1.id = v2.id;
```

Algorithm 4. Dot-production Addition (**v** = **v1'v2**)

```
Input: Vector v1, Vector v2.
Output: dot product x.
1: select sum(v1.val * v2.val) from v1, v2
2: where v1.id = v2.id;
```

Algorithm 5. Matrix-Matrix Multiplication (**M = M1 * M2**)

```
Input: Matrix M1, Matrix M2.
Output: Matrix M.
1: select M1.src, M2.dst, sum(M1.val * M2.val) from M1, M2
2: where M1.dst = M2.src group by M1.src, M2.dst;
```

One more point we have to emphasize here is that even though we express the basic operations using the terminologies matrix and vector, simply substitute the two into edges and nodes and we can apply them in graph mining.

3.2 Degree Distribution

Computing the degree of each node in a graph and visualize the distribution can give us a general idea about the information of the graph. Degree of nodes can be used as a metric to measure the centrality. Moreover, we can see that the degree distribution of most graphs follows power law. So getting this kind of information is a good start before digging deeply into the graph. The algorithm is quite simple and intuitive, so we don't have to list it here.

```
Sql for Degree Distribution
```

```
select degree, count(*) from
(select p1 as nodeid, count(*) as degree from
  ((select src as p1,dst as p2 from edge as T1)
    union all
   (select dst as p1,src as p2 from edge as T2)
  ) as T3 group by p1
) as T4 group by degree order by degree desc
```

3.3 PageRank

PageRank is a very famous algorithm that was firstly used by Google to calculate relative importance of web pages. The PageRank vector $\boldsymbol{p}$ of n web pages satisfies the following eigenvector equation:

$$\boldsymbol{p} = (cE^T + (1 - c)U)\boldsymbol{p} \qquad (1)$$

In formula (1), c is a damping factor (usually set to 0.85), E is the row-normalized adjacency matrix (source, destination), and U is a matrix with all elements set to $1/n$. We can see that the core idea of this algorithm is a matrix-vector multiplication, so we can simply apply algorithm2 and do some necessary modifications.

First, we construct a matrix M by column-normalize E^T such that each column of M sum to 1. Then the next PageRank vector is calculated by $\boldsymbol{p^{next}} = \boldsymbol{M} \times_G \boldsymbol{p^{cur}}$ where the three operations are defined as follows:

1. $combine2(m_{i,j}, v_j) = \mathrm{c} \times m_{i,j} \times v_j$
2. $combineAll_i(x_1, \ldots, x_n) = \frac{(1-c)}{n} + \sum_{j=1}^{n} x_j$
3. $\mathrm{assign}(v_i, v_{new}) = v_{new}$

```
Algorithm 6. PageRank
```

```
Input: edge table, dataset to node.
Output: pagerank score for each node.
1: initialize variables (c=0.85, maxIter = 20)
2: normalize the adjacency matrix and initialize the pagerank vector
3: for k → maxIter do
4:     $\boldsymbol{p^{next}} = (cE^T + (1 - c)U)\boldsymbol{p^{cur}}$
5: end for
```

Since the most important operation is the update of the page rank vector, we just list the sql for this operation, for the complete implementation of pagerank, please refer to the source code.

Sql for $\boldsymbol{p^{next}} = (cE^T + (1-c)U)\boldsymbol{p^{cur}}$

```
 with temp as (
  select edge.dst as dst,
    0.15/? + sum(0.85*edge.normadjm*nodes.pr) as pr
    from edge, nodes
   where edge.src=nodes.id
   group by edge.dst
 )
 update nodes set pr=temp.pr
 from temp
 where id=temp.dst
```

? indicate the number of nodes in this graph.

3.4 Random Walk with Restart

The algorithm for RWR is similar to PageRank and the three basic operations are almost the same. What makes it different from PageRank is the constant part. For the description of algorithm, simply referring to algorithm 3 is enough.

For the implementation, the structure is similar. You may refer to the source code for the complete implementation.

3.5 Connected Components

We apply HCC to find connected components in large graphs. The main idea is as follows. For each node v_i in the graph, one component id c_i^h which is the minimum node id within h hops from v_i is maintained. Initially, c_i^h of v_i is set to its own node id. For each iteration, c_i^{h+1} is set to the minimum component id of its own and its neighbors'. This kind of operations can also be viewed as a case of general matrix-vector multiplication.

$$c^{h+1} = M \times_G c^h$$

Where M is an adjacent matrix and c^h is the vector of component id. The specific definition for the three basic operations in general matrix-vector multiplication is as follows:

1. $combine2(m_{i,j}, v_j) = m_{i,j} \times v_j$
2. $combineAll_i(x_1, \dots, x_n) = MIN\{x_j \mid j = 1 \dots n\}$
3. $assign(v_i, v_{new}) = MIN(v_i, v_{new})$

The operations above can be easily translated into SQL. The definition of the algorithm is as follows.

```
Algorithm 7. Connected Components
```

```
Input: edge table, dataset to node.
Output: statistics of connected components.
1: initialize each component id of each node as the node id
2: for k → maxIter util convergence do
3:    update own component id as the minimum of self's component id
4.    and the component id of neighbors'
5: end for
```

Since the most important operation in HCC is the update operation, we just list sql for it here and you can refer to the source code for the complete implementation.

```
Sql for update in HCC
```

```
with temp as(
  select E.p2 as nid, min(E.CC*V.cc) as mini
  from
    ((select src as p1, dst as p2, CC from edge)
    union
    (select dst as p1, src as p2, CC from edge)) as E,
    nodes as V
   where E.p1=V.Id group by E.p2)
update nodes set cc=mini from temp where id=temp.nid
```

3.6 Radius

Computing precise radius for each node is not applicable in large graphs due to the size of storage. We applied the algorithm in HADI to do this job. The radii we have computed are called effective radii[2]. For a node *v* in a graph *G*, the effective radius of *v* is the 90th-percentile of all the shortest distances from *v*. We also follow HADI to apply Flajolet-Martin algorithm for counting the number of distinct elements in a multi-set. The framework for the algorithm is as follows.

```
Algorithm 8. Radii
```

```
Input: edge table, dataset to node. MaxIter and K
Output: radius for each node.
1: initialize FMBitstring for each node
2: for h → 1 to MaxIter do
3:    compute the new K bitstrings for each node and store.
4.    store the approximate number of the nodes that is reachable for each node
5:    if no bitstring has changed then
6:        break
7.    end
8: end for
9: compute effective radius for each node
```

For the implementation of radii using sql, we create two tables for storing the bitstrs of each node and one table for storing the number of nodes that can be reached by each node in each hop. Actually, we just store the immediate bitstrs for the next step, what we actually use in the final step is the number of nodes reachable for each node in each hop.

```
Sql for Radii
```

```
 1: Create three tables radii_fm1, radii_fm2, radii_n.
 2: Initiate the three tables by inserting bitstrs to radii_fm1 and radii_fm2,
    zeros to radii_n for the number of 0-hop neighbors for each node.
 3: Define a function to compute the number of nodes reachable using its bitstrs.
 4: for h → 1 to MaxIter do
 5:    with temp as (
         select
            edge.src as src,
            radii_fm1.k as k,
            bit_or(radii_fm1.bitstr) as bitstr
         from edge, radii_fm1
         where edge.dst=radii_fm1.Id
         group by edge.src, radii_fm1.k)
       update radii_fm2
       set bitstr=temp.bitstr
       from temp
       where radii_fm2.id=temp.src and radii_fm2.k=temp.k
 6:    Using the function we have defined to compute the number of nodes reach-
       able and insert into radii_n
 7:    Judge If there is any difference between radii_fm1 and radii_fm2
 8:    update radii_fm1 using the steps similar as 5-7.
 9: end for
10: compute effective radius for each node
```

3.7 Eigenvalues values

Eigenvalues of a matrix are critical properties that many important algorithms depend on, such as SVD and tensor analysis. In graph mining, eigenvalues and eigenvectors are used to calculate triangles.

We use Lanczos-NO to generate the coefficients α[1…m] and β[1…m-1]. And then we use QR decomposition iteratively to get the m eigenvalues.

Algorithm 9. Lanczos-NO (No Orthogonalization)

```
Input: Matrix $A^{n \times n}$, random n-vector b, number of steps m.
Output: coefficients α[1…m] and β[1…m-1].
1: $\beta_0 \leftarrow 0$, $v_0 \leftarrow 0$, $v_1 \leftarrow b/||b||$
2: for i = 1… m do
3:     $v \leftarrow A\, v_i$
4:     $\alpha \leftarrow v_i^T v$
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:    $v_{i+1} \leftarrow v/\beta_i$
11:end for
```

Algorithm 10. QR decomposition

```
Input: Coefficients α[1…m] and β[1…m-1].
Output: Matrix Q and R
1: formulate tri-diagonal matrix $V=[v_0,\dots,v_m]$ usingα[1…m] and β[1…m-1]
2: Q = V, Q = $[q_0,\dots,q_{m-1}]$
3: for i = 1… m-1 do
4:     $r_{ii} \leftarrow ||q_i||$
5:     $q_i \leftarrow q_i / r_{ii}$
6:     for j = i +1…n-1 do
7:         $r_{ij} = q_i * q_j$
8:         $q_j = q_j - r_{ij} * q_i$
9:     end for
10:end for
```

Just do QR decomposition and reformulate the matrix as RQ and iteratively repeat the two steps. After a number of steps, the elements on the diagonal for the matrix are eigenvalues we want to get.

From the description of the algorithms above, we can see matrix/vector operations clearly, so just make use of what we have proposed in part 3.1 and we can get an implementation of the sql version eigen-solver.

For Lanczos-NO, we don't have to list the sqls here since the pseudo-code can be easily translated into sql using the Algorithm 1-5. We just list the sqls for QR to illustrate part of what we have done in this part.

```
Sql for QR
```

```
1: for j = 1…m-1
2:  insert into t_r
   (select j, j, sqrt(sum(Power(val,2)))
    from t_q_qr where col_id=j)
3:  update t_q_qr
      set val=val/(select val from t_r where col_id=j and
       row_id=j) where col_id=j
4:    for i = j+1… m-1
5:       insert into t_r
         (select j, i, sum(tableQ1.val * tableQ2.val)
            from t_q_qr tableQ1, t_q_qr tableQ2
            where tableQ1.row_id=tableQ2.row_id and
            tableQ1.col_id=j and tableQ2.col_id=i)
6:       update t_q_qr tableQ1
           set val=val-
            (select val from t_r where col_id=i and row_id=j)*
            (select val from t_q_qr tableQ where
             tableQ.col_id=j and tableQ.row_id=tableQ1.row_id)
             where tableQ1.col_id=i
7:    end for
8: end for
```

3.8 Belief Propagation

In this section we present the approximation of Belief Propagation (BP) using linearized BP (FaBP). The FaBP algorithm approximates the solution to BP by applying the following linear system:

$$[I + aD - c'A]b_h = \Phi_h \qquad (2)$$

where I is a n * n identity matrix, D is a n * n diagonal matrix of degrees, A is a n * n symmetric adjacency matrix, and Φ_h is the n * 1 vector of the prior beliefs. What we want is the vector of the final beliefs b_h. This linear system is guaranteed to converge if we have the correct parameters a and c' according to the homophily factor h_h. The formula is as follows.

$$a = \frac{4h_h^2}{(1-4h_h^2)} \quad c' = \frac{2h_h}{(1-4\,h_h^2)} \qquad (3)$$

To compute b_h, we need to compute the inverse of the matrix $(I + aD - c'A)$, denote as $(I - W)$. Based on the following power expansion, we can have the approximation of the inverse matrix.

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

Thus the solution for the linear system is given by the formula

$$(I - W)^{-1}\Phi_h = \Phi_h + W\Phi_h + W(W\Phi_h) + \cdots$$

The framework of the algorithm is as follows.

Algorithm 11. FaBP

```
Input: edge table, dataset to node, prior beliefs vector.
Output: final belief vector.
1: pick $h_h$ to achieve convergence
2: compute the parameters a and c'
3: solve the linear system
4: if the accuracy is not sufficient
5:     prior belief vector = final belief vector
6:     solve the linear system
7: end
```

We only store the edges in the database rather than the adjacency matrix, which will be very sparse and too big. So we need join operations in our queries. Firstly, we do some initialization to get the degree of each node. Then we use 1-norm and Frobenius norm to calculate h_h. At last we solve the linear system to get the final belief. Because the initialization and normalization code are straightforward but tedious, here we only list the SQL queries to solve the linear system.

Sql for FaBP

```
1: select * from
     (select src from edge union select dst from edge) as tmp
2: for each node
      value = select sum(degree * c * coeff + coeff)
            from edge, vector
            where edge.src = nodeid and dst = vector.node
      update result set coeff = value
            where node = nodeid
3: end for
```

3.9 Count of Triangles (award winning)

We use the wedge sampling approach to count the triangles in a graph. Wedge means a path of length 2. A wedge is closed if it is part of a triangle. The main idea is that the information about triangles is usually summarized in terms of clustering coefficients. Let T denote the number of triangles in the graph and W be the number of wedges. Then estimation of the global clustering coefficient C = 3T / W.

The sampling approach to estimate C is as follows.

```
Algorithm 12. Count of triangles (award winning)
```

```
Input: edge table, dataset to node.
Output: estimation of the number of triangles.
1: compute the number of wedges in the graph, denote as W
2: for each node v in the graph
3:     compute the number of wedges centered at node v, denote as $W_v$
4:     compute the probability of picking node v. $p_v = W_v / W$
5: end for
6: select k random nodes according to probability distribution defined by $\{p_v\}$
7: compute the fraction of closed wedges as estimate of C
8: T = C * W / 3
```

The accuracy is controlled by k. Set k = $0.5\varepsilon^{-2}\ln(2/\delta)$. The algorithm outputs an estimate X for the global clustering coefficient C such that $|X - C| < \varepsilon$ with probability greater than $(1 - \delta)$.

Since we only need to know C and W to calculate the number of triangles, the SQL queries are quite simple. First we calculate the total number of wedges W. Then we select k random nodes to compute the fraction of closed wedges as estimate of C. Then T is easy to calculate.

```
Sql for count of triangles (award winning)
```

```
1: select node, count(*) as degree
   from (select src as node from edge
              union all
               select dst as node from edge) as tmp
   group by node
2: for each node
     W += degree * (degree - 1) / 2
   end for
3: for k selected random nodes
     wedge += select degree
        from wedge
        where node = nodeid
     closedWedge += select count(*)
        from edge,
             (select src as node
              from edge where dst = nodeid and src != nodeid
```

```
                  union
                  select dst as node
                 from edge where src = nodeid and dst != nodeid)
                 as p1,
                (select src as node
                  from edge where dst = nodeid and src != nodeid
                  union
                  select dst as node
                  from edge where src = nodeid and dst != nodeid)
                  as p2,
            where edge.src = p1.nodeid
                and edge.dst = p2.nodeid
                and p1.nodeid > p2.nodeid
     4: end for
     5: C = closedWedge / wedge
```

3.10 Shortest Path

In large graphs, it is impossible to load all the nodes and edges into memory and then find shortest paths. Thus we implement Dijkstra algorithm using SQL.

The basic idea of Dijkstra is greedy method. We start with a single source node. For each round, we find an unvisited node X with the lowest cost and add it to the result set. Then we consider all the nodes that can be reached. If the cost to node i meet the condition that cost(src, i) > cost(src, X) + weight(X, i), we update the cost from source node to node i. When all the nodes are visited, we can get the shortest path from the source node to each node.

The algorithm is as follows.

```
Algorithm 13. Dijkstra
```

```
Input: edge table.
Output: estimation of the number of triangles.
1: initialize the node table, denote the total number of nodes as N
2: for i = 0 to N – 1, do
3:     find the unvisited node X with the lowest cost
4:     for node j that can be reached from X in one step
5:         if cost(src, j) > cost(src, X) + weight(X, j)
6:             cost(src, j) = cost(src, X) + weight(X, j)
7:         end if
8:     end for
9: end for
```

In our SQL implementation, we maintain another node table with columns nodeid, cost and visited flag. We do the greedy method with the following SQL queries for each round.

```
Sql for Dijkstra
```

```
1: select min(cost)
   form node
   where flag = false and id != srcid
2: select id
   from node
   where id != srcid and cost = mincost
3: update node set flag = true where id = nodeid
4: select dst, weight
   from edge
   where src = nodeid
5:  for each dstNode
       cost = select cost
              from node
              where id = dstNodeid
       if cost > mincost + weight
              update node set cost = mincost + weight
              where id = dstNodeid
6:     end if
7: end for
```

4 Experiments

Our main goal is to apply our implemented methods on as many datasets as possible. We have done experiments about the 9 implemented algorithms in graph mining: Degree Distribution, Page Rank, Random Walk with Restart, Connected Component, Compute radii using 'HADI', Eigenvalue, Belief Propagation, Count of Triangles, Shortest Path.

4.1 Design of experiment

For each algorithm, we first make use of other ways to verify the validity of our implementation. Then we illustrate our observations of the results of the implemented algorithms on different datasets.

4.2 Environment

We run the algorithms on a single machine with 2 Intel Core i5-3230M 2.60GHz, 8GB memory and 500GB hard disk, running Windows 8 professional.

4.3 Data

We choose seven real world datasets to do experiments about our implemented algorithms. The datasets are listed as follows:

Advogato, 6K nodes
Subelj_Cora, 23K nodes
Web-Google, 870K nodes
RoadNet-PA, 1.1M nodes
Wikipedia Chinese, 2M nodes
Wiki-Talk, 2.4M nodes
DBPedia, 3M nodes

We choose datasets of different size in order to find whether the common phenomena can be observed regardless of the size of the graphs. And we can see that among all the datasets, 5 of them are of million-node scale.

In the following parts, we will go on illustrating the experiments for each algorithm. To save space and reduce redundancy (since the phenomena are almost the same for different large datasets), we will not list the experiment results about all the datasets for each algorithm. However, to prove that we did do experiments on all the datasets, experiment results listed for each algorithm may aim at different datasets.

4.4 Experiment Results for Degree Distribution

4.4.1 Demonstration of validity

To verify the validity, we construct a test dataset which only contains 10 nodes. We first manually computed the degree distribution and then ran our program on it. The experiment results all exactly the same

4.4.2 Experiment Results on Different Datasets

We list the experiment results for 3 of the 7 datasets (Advogato, Wikipedia-Chinese, Wiki-Talk) which are representative for the whole set of experiments on this algorithm. The plots for degree distribution are listed as follows:

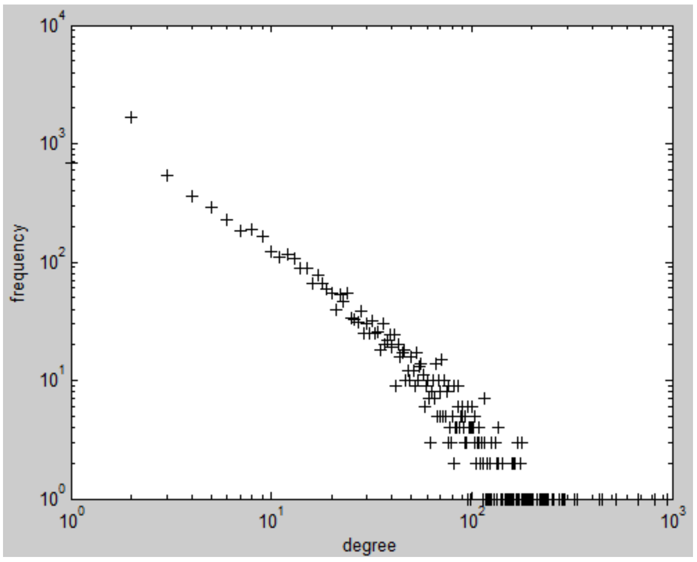


Fig 1.1 Degree Distribution for Advogato (6K nodes)

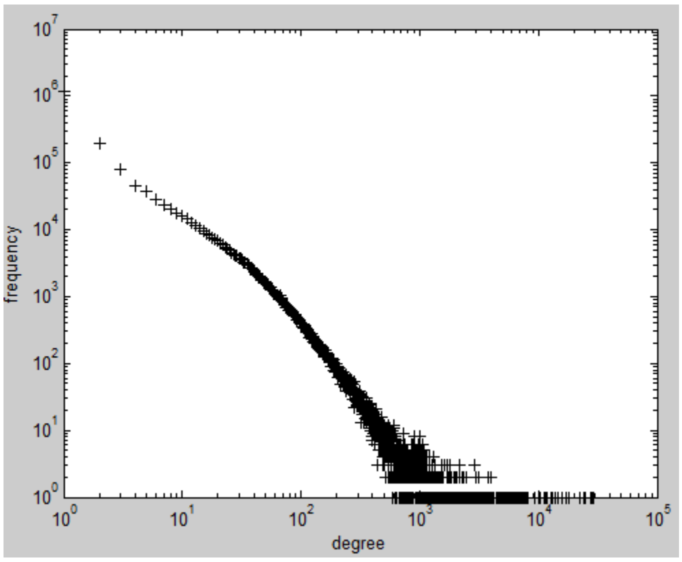


Fig 1.2 Degree Distribution for Wikipedia Chinese (2M nodes)

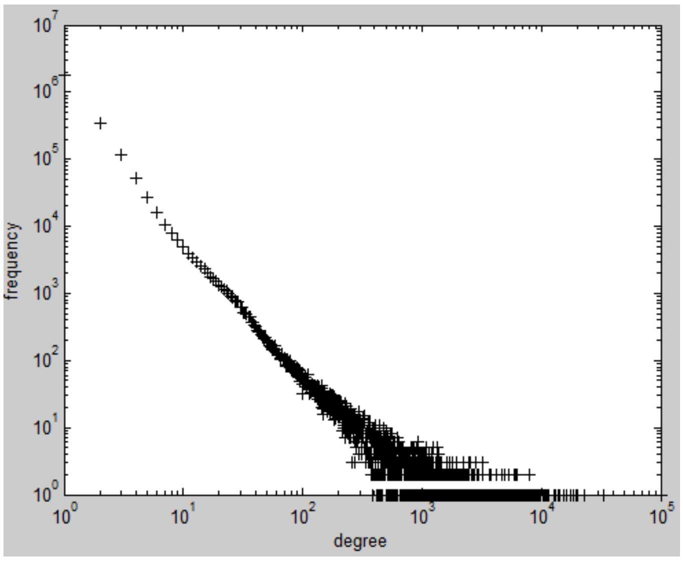


Fig 1.3 Degree Distribution for Wiki-Talk (2.4M nodes)

From the 3 plots listed above, we can see that in real world, the degree distribution for nodes follows the power law no matter how big the graph is. This point is quite intuitive to understand. We can view this point from a simple perspective, most people in real world tend to have a few connections with others, and the number of people who have more connections tends to decrease dramatically. Other kinds of graphs should have the same property.

For running time, this method is superfast which takes only a few seconds to finish on large datasets.

4.5 Experiment Results for Page Rank (RWR is omitted)

4.5.1 Demonstration of validity

To verify the validity of our implementation, we construct a small graph which contains only ten nodes, and we use matlab to compute the pagerank score for each node in this graph. The results are exactly the same. Then we run this matlab program on advogato, the results are almost the same (i.e. the rank of nodes and the page rank scores are the same as the result of our program).

4.5.2 Experiment Results on Different Datasets

We list the experiment results for 3 of the 7 datasets (Subelj_Cora, RoadNet-PA, Wiki-Talk) which are representative for the whole set of experiments on this algorithm. The plots for page rank score are listed as follows (x-axis represents the rank for each node according to pr score, y-axis is pr score accordingly):

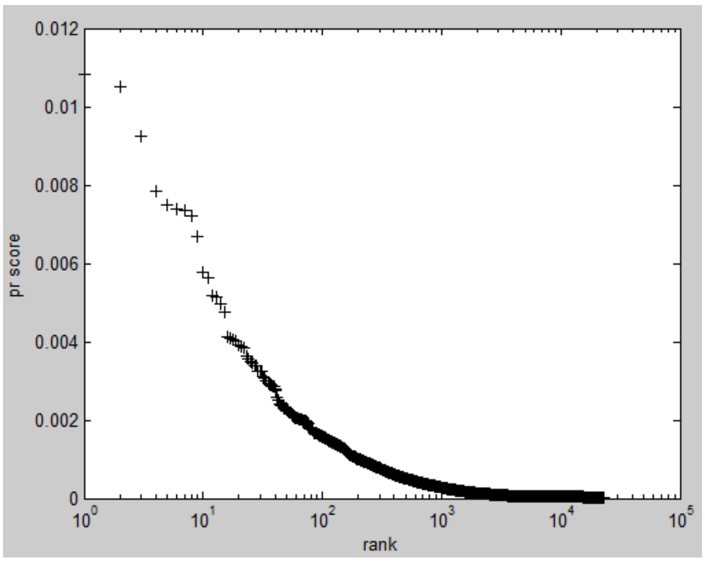


Fig 2.1 PageRank Plot for Subelj_Cora (23K nodes)

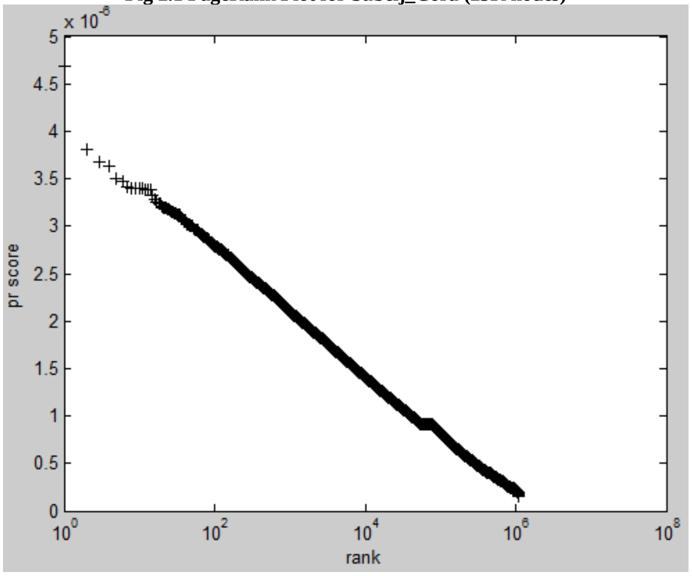


Fig 2.2 PageRank Plot for RoadNet-PA(1.2M nodes)

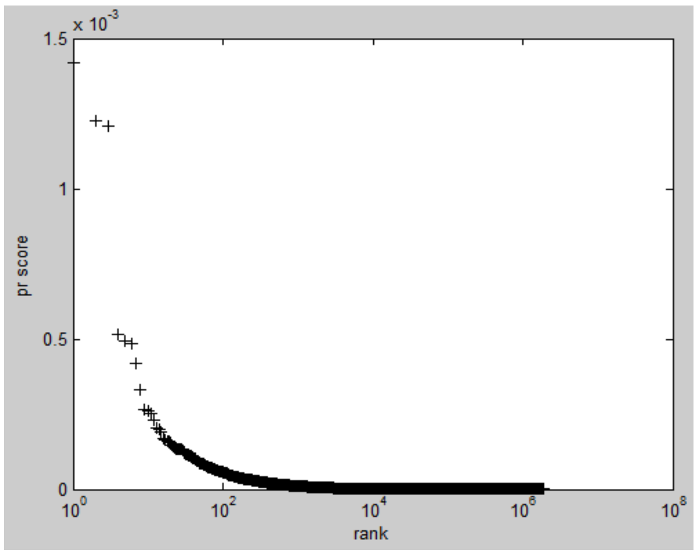


Fig 2.3 PageRank Plot for Wiki-Talk (2.4M nodes)

From the 3 plots above, we can see that pagerank score for each graph also follows power law. This point is also intuitive. In real life, the number of people who are of authority is much smaller than the number of common people. The strangest plot among the seven dataset is RoadNet-PA, we can see that the page rank scores for the nodes in this plot are not that different. This is because the plot is too sparse (the average number of edges for each node is about 2).

For running time, it takes about one hour to finish when running on the datasets that contain about one million nodes.

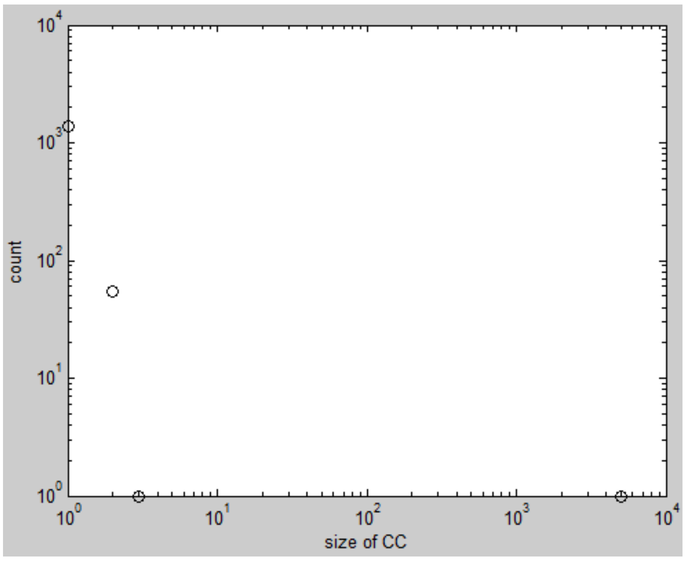
4.6 Experiment Results for Connected Components

4.6.1 Demonstration of validity

To verify the validity of our implementation, we construct a small graph which contains only ten nodes. We manually compute the connected components and compare the result with the output of our program. The results are exactly the same.

4.6.2 Experiment Results on Different Datasets

We list the experiment results for 3 of the 7 datasets (Advogato, Google-web, Wiki-Talk) which are representative for the whole set of experiments on this algorithm.



Tab 3.1 Connected Component for Advogato (6K nodes)

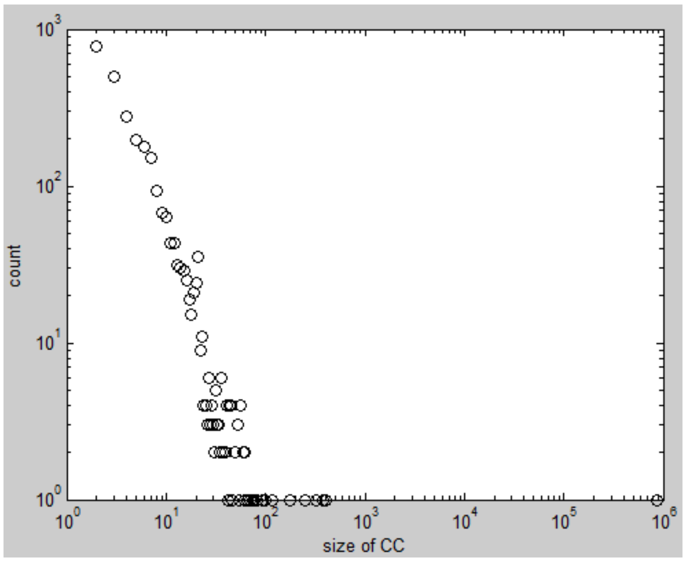


Fig 3.2 Connected Component for Google-web (870K nodes)

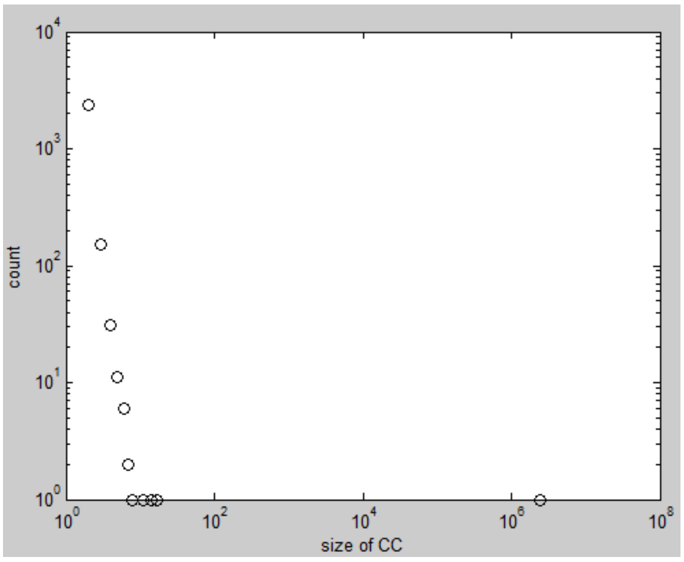


Fig 3.3 Connected Component for Wiki-Talk (2.4M nodes)

From the 3 plots above, we can see that there is always a very huge connected component and some small connected components in each graph regardless the size of the graph. We can think about this point in an intuitive way. In one party, there should be one group which contains most of the people in this party, there can also be some small groups that have their own aim although they are claimed to be in this party. Sometimes, there might be more small connected components for some graphs compared with others, just as figure 3.2 tells us. But the general principle should be the same.

For running time, it takes hours to finish on datasets which are of million-node scale.

4.7 Experiment Results for Radii (HADI)

4.7.1 Demonstration of validity

We manually construct a small graph which contains only 20 nodes. Since the radii we computed are effective radii which are not the exact radii, the two results are not exactly the same, but the tendencies are almost the same.

4.7.2 Experiment Results on Different Datasets

We list the experiment results for 3 of the 7 datasets (Advogato, Google-web, Wiki-Talk) which are representative for the whole set of experiments on this algorithm.

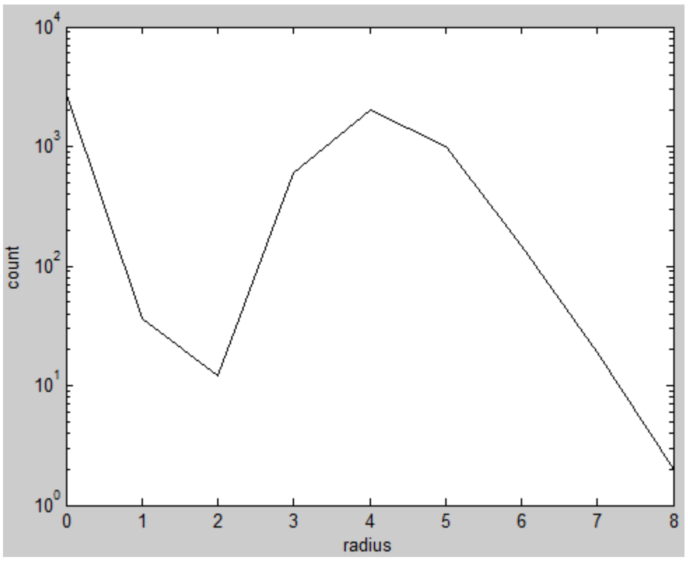


Fig 4.1 Radii for Advogato(6K nodes)

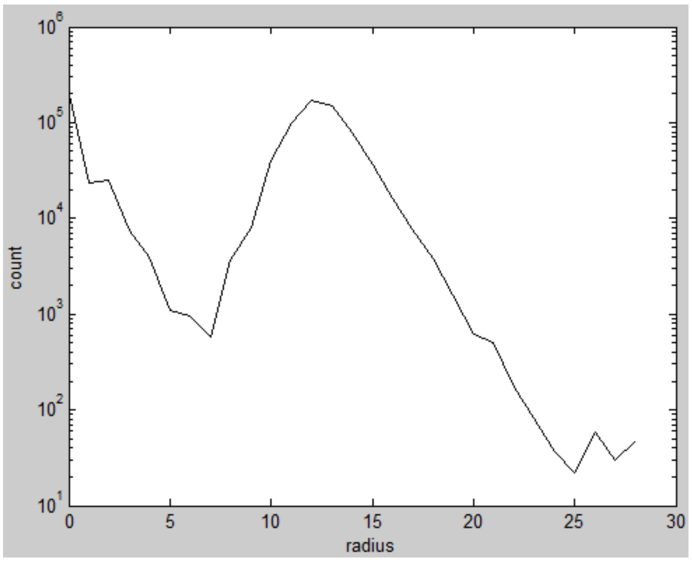


Fig 4.2 Radii for Google-web(870K nodes)

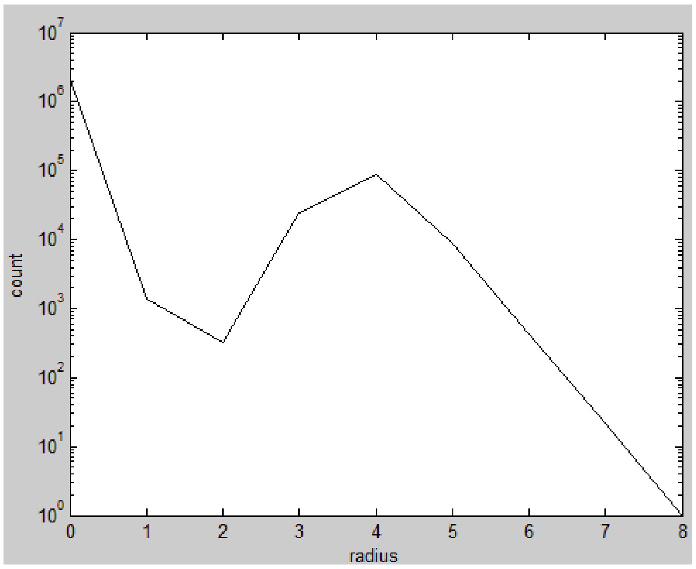


Fig 4.3 Radii for Wiki-Talk(2.4M nodes)

From the 3 plots above, we can see that the general tendencies for all the graphs are almost the same no matter how big the graph is. As the radius increase, the number of nodes that has this radius first decreases, then increases and then decreases. This point is not that intuitive before we take a deep thinking. Most nodes tend to have radii that are not too large or too small. The number of nodes which have larger radii or smaller radii is much smaller. This phenomenon is kind of like Six Degree of Separation and Small World Phenomenon.

It takes hours for this algorithm to finish on datasets that are of million-node scale.

4.8 Experiment Results for Eigenvalue

4.8.1 Demonstration of validity

This time, first we construct a tri-diagonal matrix and use the eigen function in matlab to compute the eigenvalues for this matrix, and then we test our program for the loop of QR decomposition, we see that when the number of loop is big enough, the eigenvalues for both of the results are almost the same.

For the part of Lanczos-NO, we implemented the matlab version of Lanczos-NO and tested it on the advogato dataset. We see the alpha and beta coefficients are the same when the initial values are the same.

In this way, we demonstrate the validity of our program.

4.8.2 Experiment Results on Different Datasets

We list the experiment results for 3 of the 7 datasets (Subelj_Cora, Google-web, RoadNet-PA) which are representative for the whole set of experiments on this algorithm.

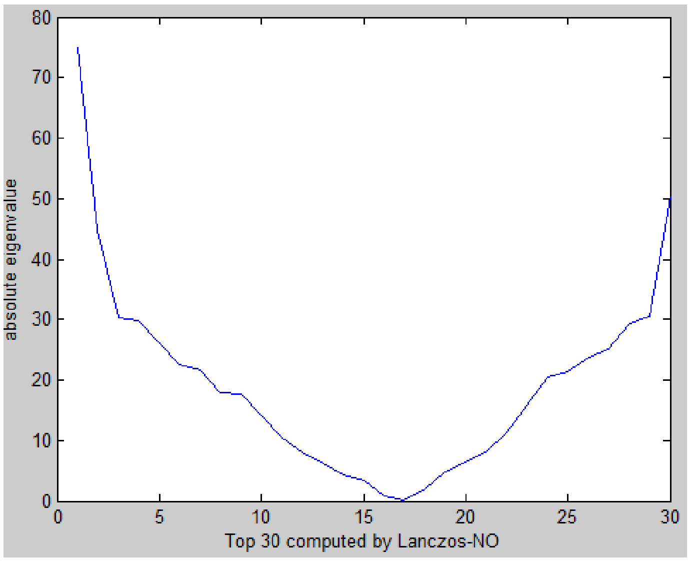


Fig 5.1 Eigenvalues for Subelj_Cora(23K nodes)

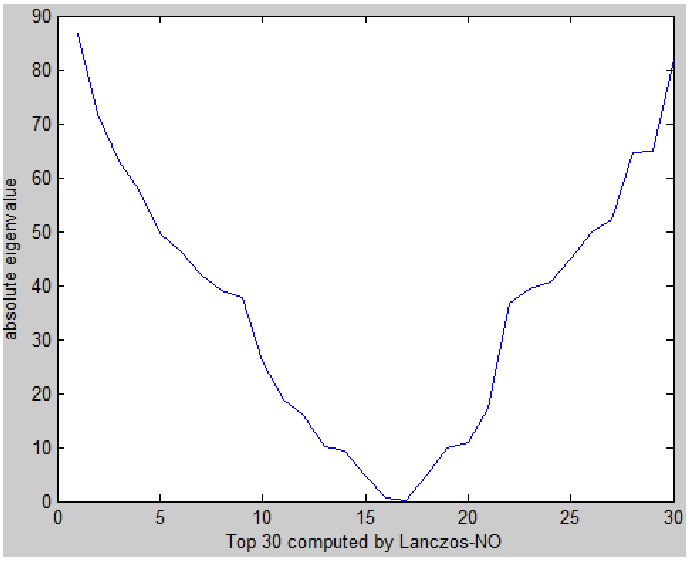


Fig 5.2 Eigenvalues for Google-web(870 K nodes)

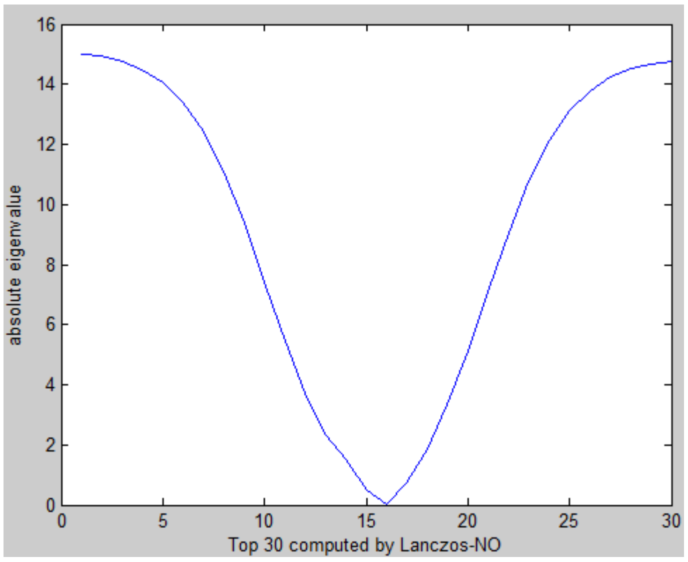


Fig 5.3 Eigenvalues for RoadNet-PA(1.2 M nodes)

We can see that Lanczos-NO is an effective way to compute the extreme eigenvalues for large graphs. The absolute eigenvalue graphs are all bowl like. Even though I don't know why it should be like this, I guess it is because the real world graph are dominated by a small number of eigenvalues that are larger than average. Or we shouldn't get this kind of plot.

For running time, it takes hours for this algorithm to finish on million-node scale graph.

4.9 Experiment Results for Belief Propagation

4.9.1 Demonstration of validity

To verify the correctness of our algorithm, we apply Advogato and write a Matlab program to do the matrix multiplication. Since the FaBP algorithm solves the linear system with approximation, the final results are not exactly the same but quite similar.

4.9.2 Experiment Results on Different Datasets

We list our experiment results for the following three dataset, Advogato, Web-Google and RoadNet-PA. We use the belief rank as the x-axis and the belief value as the y-axis. The plots are as follows.

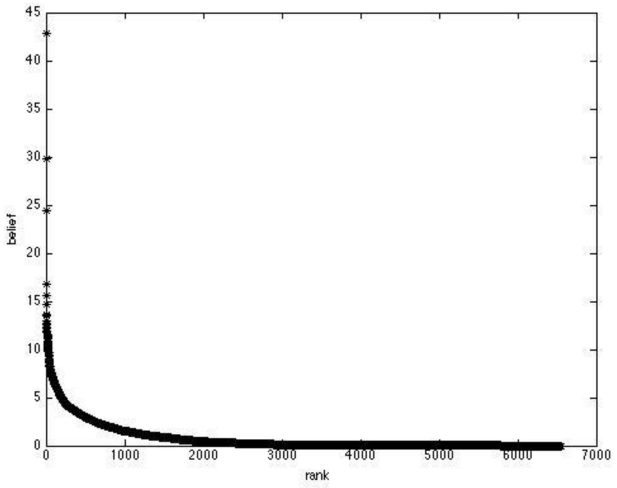


Fig 6.1 Belief Propagation for Advogato(6 K nodes)

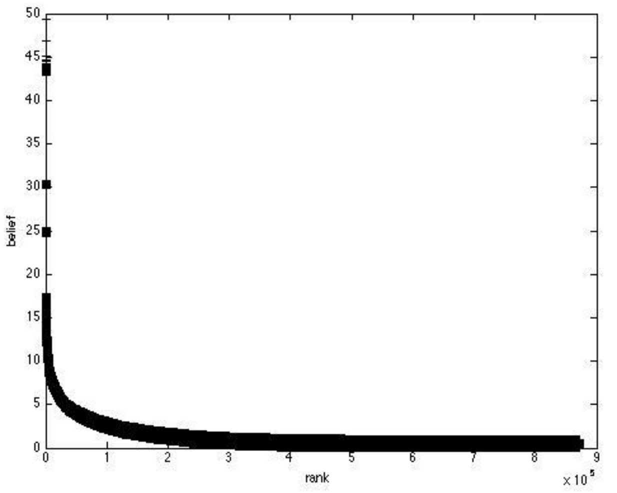


Fig 6.2 Belief Propagation for Web-Google(870 K nodes)

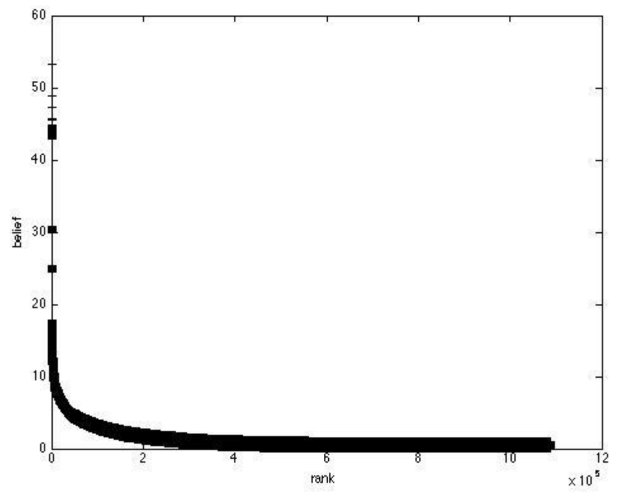


Fig 6.3 Belief Propagation for RoadNet-PA(1.2 M nodes)

From the three plots above, we can see that the belief distribution follows power law. The nodes with very high belief values are very few, while the nodes with nearly zero belief value consist of the biggest portion of the dataset. So in graph mining, we can use belief propagation to find out the outliers with very high belief value. These nodes tend to be of great importance.

For the running time, it takes up to several hours to finish on million-scale datasets.

4.10 Experiment Results for Count of Triangles

4.10.1 Demonstration of validity

We use a small dataset with about 100 nodes. Since the wedge algorithm is based on sampling, we run our program 50 times to get the average number of triangles in the dataset as a more precise approximation. We also use Matlab to calculate the exact number of triangles in this graph. The results are almost the same.

4.10.2 Experiment Results on Different Datasets

The following plots show the results for five dataset, Advogato, Subelj_Cora, Web-Google, RoadNet-PA and Wikipedia Chinese. The first plot shows the correlation between the number of the edges and the number of triangles (larger number of edges means larger datasets). The second plot shows how the different constraint k affects the estimate of the triangle number. (Since we only test 5 different k, there are only 5 points in that plot, this plot is aimed at Advogato)

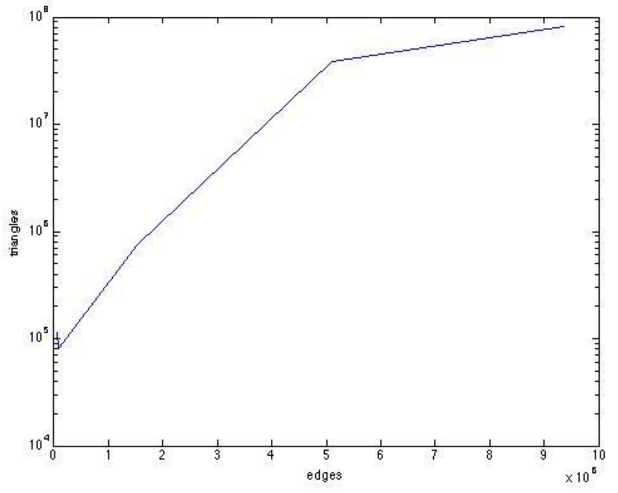


Fig 7.1 Number of Triangles for 5 datasets

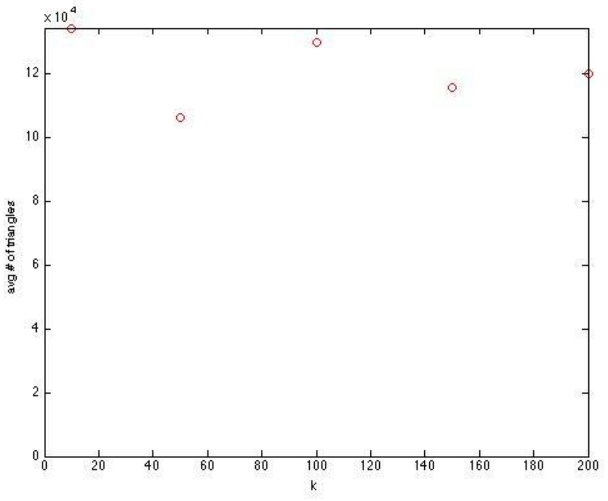


Fig 7.2 Number of Triangles using different parameter k

From Fig 7.1, we can see that when the number of edges grows, the total number of triangles grows. But we can see there is a low slope in the right. We think it is because that the RoadNet-PA dataset is very sparse. Though there are millions of edges, the average degree of each node is about two.

From Fig 7.2, when we use greater k to improve the accuracy of our estimation, the fluctuation is smaller. Also, the number of triangles seems to converge to a certain value. So we can say that with greater k, the accuracy of estimation will be better.

Wedge approximation method is quite fast. It takes about 1.5 hours to finish even on million-scale datasets.

4.11 Experiment Results for Shortest Path (Dijkstra)

4.11.1 Demonstration of validity

We construct a small dataset with 10 nodes to verify the correctness of our program. The results are exactly the same with the results we calculate manually. We also implement the algorithm in C. The results turn out to be correct.

4.11.2 Experiment Results on Different Datasets

The following plots show the results for Advogato, Web-Google and RoadNet-PA datasets. We use the node with the smallest node id as the single source node. The x-axis is the cost of the shortest path while the y-axis is the frequency for a certain cost. We exclude the nodes that cannot be reached from the source node.

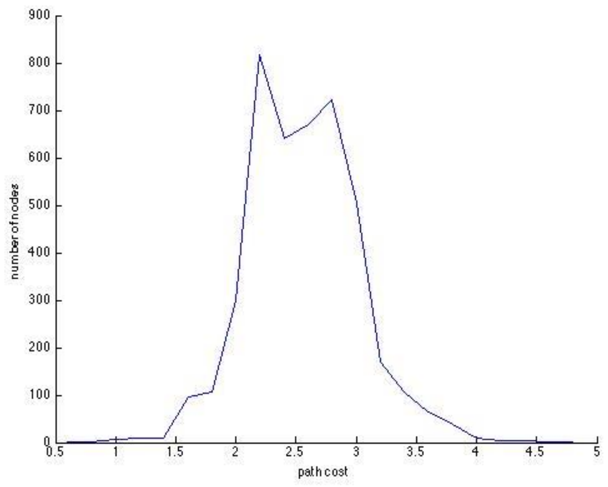


Fig 8.1 Shortest Path for Advogato(6 K nodes)

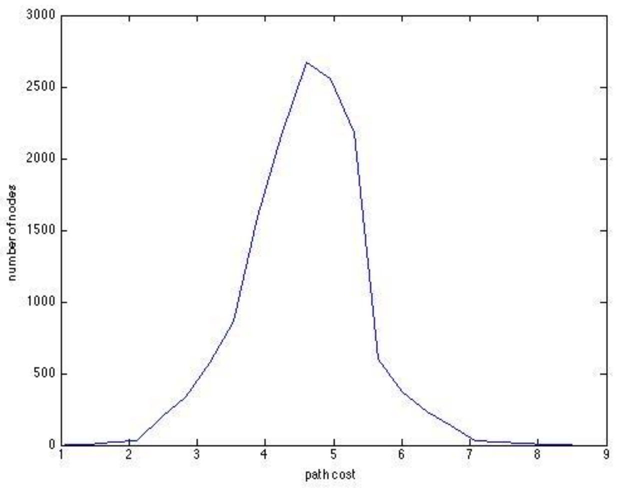


Fig 8.2 Shortest Path for Web-Google (870 K nodes)

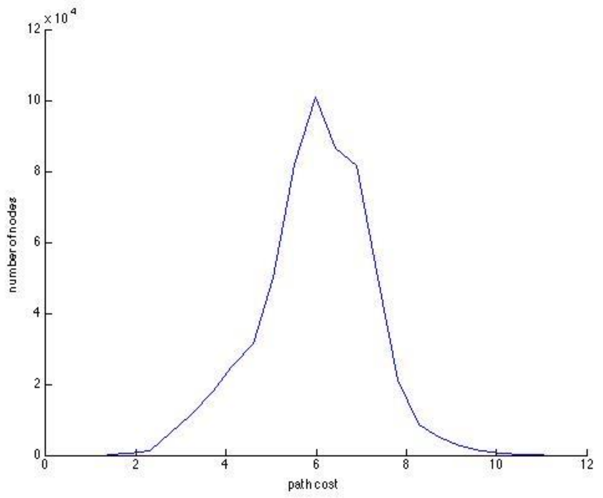


Fig 8.3 Shortest Path for RoadNet-PA(1.2 M nodes)

As we can see from the figures above, the distribution follows Gaussian distribution. Intuitively, in the real-world social network, most of our friends are not that familiar with us, while our close friends are few. Also, we can see that when the size of the graph grows, the medium path cost grows. It is intuitively right because it takes more hops to reach a far-away node in a huge graph.

For the running time, it takes several minutes to fininsh on small datasets like Adv ogato, while several hours on million-scale datasets.

5 Conclusion and Future Work

We have implemented nine basic algorithms in graph mining using RDBMS. We see that many algorithms can be implemented using RDBMS beautifully and succinctly. However, there are still some bottlenecks for our implementation such as disk transfer rate. And we see using sql is very tricky since a small change may lead to big difference in efficiency.

Future work include further optimization of the RDBMS engine and avoiding the too many update operations

References

1. U. Kang, Charalampos E. Tsourakakis and Christos Faloutsos. PEGASUS: Mining Peta-Scale Graphs. In ICDM, 2009.
2. U. Kang and Charalampos E. Tsourakakis. HADI: Mining Radii of Large Graph. In TKDD, 2011.
3. U. Kang, Brendan Meeder, and Christos Faloutsos. Spectral Analysis for Billion-Scale Graphs: Discoveries and Implementation
4. U. Kang, Hanghang Tong, Jimeng Sun, ChingYung Lin and Christos Faloutsos. GBASE: an efficient analysis platform for large graphs. In VLDB Journal 2012.
5. U. Kang, Duen Horng Chau and Christos Faloutsos. Mining Large Graphs: Algorithms, Inference, and Discoveries. In ICDE, 2011.
6. Charalampos E. Tsourakakis. Counting Triangles in Real-World Networks using Projections. In Knowledge and Information Systems.

A Appendix

A.1 Labor Division

The team performed the following tasks

Degree Distribution of the nodes. [Kuo]
PageRank score of the nodes. [Kuo]
Connected Components in the graph. [Kuo]
Radius, for every node. Use the Martin-Flajolet method in 'HADI'. [Kuo]
Eigenvalues/sigular values: Use the method in HEIGEN. [Kuo]
Extra task for Kuo (RWR).
Belief Propagation - use the binary, "Fast" belief propagation. [Yuning]
Count of Triangles. [Yuning]
Extra task for Yuning(Dijkstra).
Adjusting some algorithms so that they can be applied on directed graphs. [All]
Experiments on seven real world dataset (including seven large datasets). [All]
Broad-spectrum graph mining. [All]
Try in some other directions to do innnovation. [All]

A.2 Full disclosure wrt dissertion/projects

Kuo Liu: He is performing the first five tasks with his extra task, collecting datasets and doing experiments on these datasets. He will try to implement the project in other directions to do some innovation, such as to extend the algorithms for graphs other than undirected graphs and so on.
Yuning He: He is performing the last three tasks with his extra task, collecting datasets and doing experiments. He will also try to implement the project in other directions to do some innovation.

Contents