15-826 Project Phase 3 Report

Jiajun Wang
jiajunwa@andrew.cmu.edu

San-Chuan Hung
sanchuah@andrew.cmu.edu

November 26, 2014

1 Introduction

Graph Mining is an important research field focusing on mining interesting patterns in graph data. To help users mine graph data, Nijith Jacob and Sharif Doghmi developed a SQL-based tool: GraphMiner. In this project, we enhanced GraphMiner, analyzed graph data, and provided friendly scripts for using.

We strengthened GraphMiner in two ways: First, we added K-core detection algorithm into GraphMiner. Second, we conducted experiments of indexing to accelerate computing time, and found out a good setting for indexing.

Moreover, we used revised GraphMiner to analyze 20 graphs in vary dimensions: degree distribution, triangle counting, weakly connected component, pagerank, eigenvalues, and K-core. We found some general patterns across these 20 graphs. Also, we discovered some strange patterns in some of graphs.

Besides, we provided an user-friendly script to help users to analyze data: "easy.sh" and a script for unti-test in makefile, "make test". Please check the code in our package.

This report is organized as follows: in section 2, we summarized related work. In section 3, we described our K-core algorithm implementation with details. In section 4, we provided some sample graphs for unit test. In section 5, we described how we did the indexing experiments. Finally, in section 6, we used revised GraphMiner to analyze 20 graphs.

2 Literature Survey

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

2.1 Papers read by Jiajun Wang

The first paper was by Kang et al. [5]

- *Main idea*: This paper proposes a graph mining package named PEGASUS, which performs multiple graph mining tasks based on Hadoop platform. The core innovation of PEGASUS improves graph mining performance by using MapReduce. In order to achieve this, the paper firstly summarizes different mining operations and models these operations into a matrix-vector multiplication expression, which is called "Generalized Iterative Matrix-Vector multiplication" (GIM-V). GIM-V focuses on typical datasets consistent with node and edge data. It requires 3 processing steps: "combine2", "combineAll", "assign". On one side, these 3 steps are mapped to matrix-vector multiplication. On the other side, these 3 steps can be mapped to different MapReduce stages. According to the definition, we can find that GIM-V steps are strongly related with map-reduce processes. Particularly, "combine2" can be done using a map-reduce process and then the output is sent to another map-reduce process, which implement "combineAll"/"assign" functions. The basic version of this implementation is called "GIM-V Base". Secondly, the paper discusses how to improve the basic version of GIM-V. Following methods are discussed: 1. Block Multiplication (Perform GIM-V on non-zero blocks, which forces co-locating and shorten sorting time). 2. Clustered Edges (Preprocessing for clustering edge files for better performance). 3. Diagonal Block Iteration (Preprocessing diagonal blocks to reduce iteration). 4. Node Renumbering (Make the minimum node located at the center of the graph to reduce iteration). Thirdly, the paper analyzes and evaluates the performance of GIM-V with different improving methods. The experiments compared different scenarios from 2 aspects. The first one is scalability. We can see from the chart that the run time of GIM-V is decreased while the number of machines is increased. The second one is comparing between different GIM-V versions. We can see that basically, GIM-V BL-CL out performs other methods. Finally, GIM-V is applied into real world tasks to check usability.
- *Shortcomings*: For future research, I think besides finding more ways to improve the algorithm, this package can also be enhanced at the system layer. For handling very large graph data more efficiently, we can customize Hadoop system so as to optimize disk IO and cache.

The second paper was by Kang et al. [3]

- *Main idea*: This paper introduces GBASE, which supports both storing and querying for large-scale graph data. The main focuses of GBASE are faster query while using less storage space. Basically, GBASE reaches this goal by using parallel processing and distributed compressed storage based on MapReduce framework. Firstly, in order to save graph data more efficiently, GBASE requires the data to be compressed. By assuming the data set has homogeneous block, the paper proposes block based compression algorithms?which ignore empty blocks and encode informative blocks. The

interesting thing here is that although compressing and uncompressing data cost extra time, this method does not affect overall query response time. Because compressed data is much smaller than original one. Another fact that enhances query is the way GBASE stores data. After compression, the data is stored following "Grid placement". Which maximizes data localization and minimize the accessed files during query. Secondly, the paper analyzes nature of supported graph queries. By leveraging MapReduce framework, GBASE implements efficient scalable graph queries. Finally, GBASE is evaluated using large graph data. The results of the experiments show two things. Initially, compression plays very important role in both storage saving and query performance enhancement. We see the data storage consumption is reduced 43x. Next, MapReduce framework does provide scalability. Big data set is processed faster when increase machines.

- *Shortcomings*: One problem of this paper may be the assumption of homogeneous block. For a random graph data, block compression may not work well. But on the other hand, this fact means GBASE can handle graph that has homogeneous blocks very effectively. Another point for future work in my opinion is using cache to accelerate compression/decompression process to further improve the query response time.

The third paper was by Kang et al. [4]

- *Main idea*:This paper introduces a new graph mining algorithm package named HADI, which is designed to estimate diameter and radius of large graphs. The key points of HADI are 1. approximation instead of calculate the exact value, 2. The algorithm is scalable and could be applied to MapReduce framework. These innovations optimize the algorithm performance. So HADI is able to handle massive data in reasonable time. In the first part, the author defines effective radius and diameter for approximation. Basically, the algorithm will take 90% as a threshold. And the result can still keep the important information about the graph. Then a scalable algorithm based on Flajolet-Martin algorithm is proposed. This algorithm requires O(n log n) space so it cannot be executed in a single machine for big data input. Subsequently, the author introduces HADI algorithm. It is a disk-based algorithm so as to deal with large graph. Initially, a 2 stages algorithm HADI-naive is illustrated. Naive method divides previous algorithm in to a 2 phases MapReduce process. But it is not efficient enough. Then improved version of HADI algorithm is proposed. The basic idea is using an extra MapReduce phase to illuminate the copying unnecessary bitstrings. Moreover, 2 more optimizations are applied for handling block operations and compress the input size. As a result, HADI achieves O(d(m+n)/M log (m+n)/M) time complexity and O((m+n) log n) space complexity (n for nodes number, m for edges number and d for diameter). Thirdly, the paper reports lot of experiment results to prove the stability and efficiency of HADI. We can see that using more nodes will shorten the running time of the algorithm. Although it is not linear related due to the overhead of the framework, the chart is persuasive to show the scalability. On the other hand, comparing between different optimized versions show that the final version works much better than the naive one. Finally, the author applied HADI on real world data and found

interesting result. It demonstrates the power of HADI and validates effective radius and diameter definition.

- *Shortcomings*: For the future work, besides including more graph mining operations and supporting larger data size, I think the author can also try to leverage some in-memory distributed database for overcoming the shortage of disk-based.

2.2 Papers read by San-Chuan Hung

The first paper was by Alvarez-Hamelina et al. [1]

- *Main idea*: The work proposed a large-scale graph visualization algorithm based on k-core. K-core means a sub-graph where the degrees of the nodes in the sub-graph are equal or higher than k. The visualization algorithm plots the points in the highest k-core to the center of the graph, and then plots the points in the second-highest k-core around previous highest k-core points, and so on.
- *Shortcomings*:
  - The visualization algorithm may not be able to present the importance of "weak links," because the degree of the nodes in weak links may be not large enough to be shown in k-core algorithm. For example, for a bridge node whose degree is 2 but linking two large components, the node will not be focused because it?s coreness is low.
  - The links are not weighted in the proposed algorithm. In some cases, the weights on links matter (for example, the intersect holding networks of companies); however, the proposed visualization algorithm dose not utilize link weights.
  - In the proposed algorithm, it just random sampled the links to present. As it was mentioned in the paper, edge reduction can be more sophisticated to emphasize the connections between groups or nodes. For example, it can just present important (high betweenness) edges.

The second paper was by Kang et al. [2]

- *Main idea*: The work proposed a framework for graph mining called HEIGEN to utilize Hadoop to do spectral analysis on large-scale data. Especially, HEIGEN accelerates the efficiency by choosing right part of operations to parallelize and reducing network traffic by dividing adjacency matrix into blocks. HEGIEN is useful for finding clusters and calculating triangles in large graphs.
- *Shortcomings*:
  - Maybe HEIGEN can be shifted to memory-based map-reduce framework, like Spark, which is more efficient than Hadoop.

The third paper was by Koutra et al. [6]

- *Main idea*: The work compares different label propagation models, predicting the label of nodes in a graph by propagating known node labels through links, like Random

Walk with Restarts, Semi-Supervised Learning and Belief Propagation(BP). It also proposed FABP model, implementing modified BP model in Hadoop framework, which is scalable, as accurate as BP model, and faster than BP model.

- *Shortcomings*:
  - The experiments of FABP in this paper are mainly two-classes label classification (AI papers vs. NOT AI papers, Educational websites vs. Adult websites). I think future experiments can test FABP on multiple-classes label cases, like predicting websites is Adult/Educational/Financial.

3 K-cores Manual and Implementation

K-core is the groups in which the members linked into more than K other members. As K increases, the K-core groups become more highly connected, which are usually the cores of the graph.

We provided K-core detection in gm_main.py. User can set –k to denote the parameter K in gm_main.py. After executing gm_main.py, it will generate K-core result to the output directory.

To detect K-core, we developed an iterative algorithm to detect K-core groups. We also modified some origin code in gm_main.py, making some methods can be assigned specific source table and dest table, so our k_core function can utilize existing functions.

Algorithm 1 K-core detection

Input: G: A graph G = V, E, where V is the set of nodes, and E is the set of edges. K: The parameter of K-core. A member in a K-core links to at least K other same group members.

Output: The node ids with corresponding K-core id

1: Initializing a temp graph as the original graph G
2: Calculating the degree of temp graph
3: Removing the nodes whose degree less than K
4: If there are nodes deleted in the previous step, go to Step 2; otherwise, continue to the next step.
5: Applying weakly connected component algorithms to temp graph to find K-core
6: **return** K-core groups;

The main code of k-cores algorithm is shown below.

```
def get_row_count(table):
    cur = db_conn.cursor()
    cur.execute("SELECT COUNT(*) from %s" % table)
    count = cur.fetchone()[0]
    cur.close()
    return count

def k_core(k=5):
    cur = db_conn.cursor()

    isFinished = False

    last_node_num = -1

    temp_link_table = "TMP_GM_TABLE_UNDIRECT"
    temp_link_table_2 = "TMP_GM_TABLE_UNDIRECT_2"
    temp_degree_table = "TMP_GM_NODE_DEGREES"

    # copy links from GM_TABLE_UNDIRECT to temp_link_table
    gm_to_undirected(
        gm_table_name=GM_TABLE,
        gm_table_undirect_name=temp_link_table
    )

    gm_sql_table_drop_create(
        db_conn,
        temp_link_table_2,
        "src_id integer, dst_id integer, weight real"
    )

    while not isFinished:
        # calculate current degree for each node
        gm_node_degrees(
            gm_table_name=temp_link_table,
            dest_table_name=temp_degree_table)

        # remove the node whose degree is under k
        print "before delete nodes: %d " % get_row_count(temp_degree_table)
        cur.execute("DELETE FROM %s" %(temp_degree_table) +
                " WHERE in_degree < %d" % k
        )
```

```
db_conn.commit()
print "after delete nodes: %d " % get_row_count(temp_degree_table)


print "before delete links: %d " % get_row_count(temp_link_table)
gm_sql_table_drop_create(
    db_conn,
    temp_link_table_2,
    "src_id integer, dst_id integer, weight real"
)

cur.execute("INSERT INTO %s (src_id , dst_id, weight) "
                        %(temp_link_table_2) +
            " SELECT src_id, dst_id, weight FROM %s "
            %(temp_link_table) +
            " LEFT JOIN %s as ANode ON %s.src_id = ANode.node_id"
            %(temp_degree_table, temp_link_table) +
            " LEFT JOIN %s as BNode ON %s.dst_id = BNode.node_id"
            %(temp_degree_table, temp_link_table) +
            " WHERE ANode.node_id is NOT NULL AND
            BNode.node_id is NOT NULL"
)

gm_sql_table_drop(db_conn, temp_link_table)

gm_sql_create_and_insert(
    db_conn,
    temp_link_table,
    temp_link_table_2,
    "src_id integer, dst_id integer, weight real",
    "src_id, dst_id, weight",
    "src_id, dst_id, weight"
)

db_conn.commit()
print "after delete links: %d " % get_row_count(temp_link_table)

# check if the number of nodes is changed. If no, break
current_node_num = get_row_count(temp_degree_table)
print "current_node_num = %d " % current_node_num

if(current_node_num == last_node_num):
```

```
            isFinished = True
            break
        else:
            last_node_num = current_node_num

    if current_node_num > 0:
        # component detection
        gm_connected_components(
            num_nodes=current_node_num,
            con_comp_table_name=GM_K_CORE,
            node_table_name=temp_degree_table,
            link_table_name=temp_link_table
        )
    else:
        gm_sql_table_drop_create(
            db_conn=db_conn,
            table_name=GM_K_CORE,
            create_sql_cols="node_id integer, component_id integer"
        )

    cur.execute("DROP TABLE %s" % temp_link_table_2)
    cur.execute("DROP TABLE %s" % temp_link_table)
    cur.execute("DROP TABLE %s" % temp_degree_table)

    db_conn.commit()
    cur.close()
```

4 Unit Test

Please modify gm_params.py for updating database certification and execute "make test" to run unit test. "make plot" will do unit test and then plot the result. All the result will be written into the ./output folder.

5 Indexing Experiment

5.1 Indexing Experiment

In the experiment, we treated all graphs as direct ones. So "as-skitter.ungraph-75000.txt" is extended to a direct graph.
Tested graphs include: p2p-Gnutella31.txt, as-skitter.75000.txt, ca-AstroPh.txt, email-EuAll.txt,

cit-HepTh.txt.

5.1.1 Degree distribution

Gm Node Degrees
The algorithm aggregates the count of dist nodes and src nodes in GM_TABLE for counting in degree and out degree.
We tried to add hash index and btree index on dist_id and src_id. The result shows that indices do not improve the performance. Actually the overhead of building the index makes the total running time longer.
One explanation of this result is that "group by" goes through all data. Index does not optimize the total sql running time in this case.

Index	p2p-Gnutella31.txt	as-skitter.75000.txt	ca-AstroPh.txt	email-EuAll.txt	cit-HepTh.txt
None	1.577036858	3.187309027	1.678581953	3.384658098	3.223124027
hash	2.431274891	4.133288145	4.991292953	10.31514502	3.931537151
btree	2.418382883	4.755445004	2.523052931	5.85737586	3.046495914

Gm Degree Distribution
The algorithm aggregates the count of out_degree and in_degree in GM_NODE_DEGREES for counting degree distribution.
We tried to build indices on out_degree and in_degree for testing whether these indices can accelerate the group by. Notice that the original count is very fast, so I change the code. The group by sql will run 100 times for time estimating.
Basically we have 2 columns in the scope: in_degree and out_degree. We tried: 1. hash index on both columns; 2. btree index on both columns; 3. joint btree index on the columns; 4. joint btree index plus separate btree indices on both columns. The result shows that indices do not help the sql running.
The reason is the same as "gm_node_degrees". If the sql needs to go through the whole table anyway, indices do not improve the performance.

Index	p2p-Gnutella31.txt	as-skitter.75000.txt	ca-AstroPh.txt	email-EuAll.txt	cit-HepTh.txt
None	16.55248189	18.40389895	12.65532494	35.27958608	9.814982176
hash	14.32086205	20.65242195	9.01839304	50.09777999	9.146636009
btree	13.04618001	17.1775279	9.203239918	30.25537395	11.63468695
joint btree	12.15740585	17.67118001	11.81989694	31.76970196	9.897273064
all btree	12.51487803	12.83897305	12.14183187	31.41780305	8.73434186

5.1.2 PageRank

Index of columns in "WHERE" condition can help MySQL speed up value comparison in join operation. Therefore, we found that there are 5 possible positions to add in-

dex: GM_Table.src_id, norm_table.src_id, GM_PAGERANK.node_id, offset_stable.node_id, and next_table.node_id.
According to experiment result, we found that adding B-tree index on GM_PAGERANK.node_id improved the performance best. We also tried many index combinations, but the performance did not increase. Therefore, we decided to add B-tree index on index on GM_PAGERANK.node_id for Pagerank algorithm.

Index Type	Index column	p2p-Gnutella31	ca-AstroPh	email-EuAll	cit-HepTh	as-skitter.75000	Improvement
No Index		5.253519	4.519811	41.935326	12.899988	17.064826	(baseline)
Btree	GM_Table.src_id	3.482836	4.240341	28.786316	5.008184	12.756178	31.533%
Hash	GM_Table.src_id	5.645929	5.414358	33.585078	6.153968	14.20149	12.345%
Btree	norm_table.src_id	3.725514	7.023653	34.940364	5.243184	11.328331	16.667%
Hash	norm_table.src_id	4.831202	8.649363	37.06228	9.590399	11.593534	-2.797%
Btree	GM_PAGERANK.node_id	2.834585	4.510517	28.573334	5.31056	12.19424	33.097%
Hash	GM_PAGERANK.node_id	2.998954	6.758051	33.857276	5.976427	10.206583	21.303%
Btree	offset_stable.node_id	3.235787	5.560484	32.713935	9.890352	14.586771	15.044%
Hash	offset_stable.node_id	6.353038	6.031949	35.412756	4.819224	14.377134	7.912%
Btree	next_table.node_id	3.348404	7.015845	57.072034	4.326479	8.313848	12.537%
Hash	next_table.node_id	2.320558	4.440724	27.453469	4.527795	10.225985	39.417%

5.1.3 Weakly connected components

Firstly, the sql needs to update the component ids by comparing node ids based on link_table (GM_TABLE_UNDIRECT) in a loop. The component id is retrieved from the minimum node_id. So btree index should help.
Secondly, vector different is calculated based on node_id and component_id. So hash index on node_id column should help because there is a node_id = component_id condition in the sql.
Initially, we tried btree index on GM_CON_COMP.component_id. It does improve the performance. Then we add hash index on GM_CON_COMP.node_id. It turns out that the performance is improved again. After that, we tried to add hash index on the temp table and GM_TABLE_UNDIRECT table's columns. But the enhancement is not obvious.
So the 2 indices do work is btree on GM_CON_COMP.component_id and hash on GM_CON_COMP.node_id. For the first one, it mainly improves MAX() function. For the second one, it improves the "where" condition in sqls. After these 2 are added, other additional indices only increasing overhead instead of shorten the running time.

Index	p2p-Gnutella31.txt	as-skitter.75000.txt	ca-AstroPh.txt	email-EuAll.txt	cit-HepTh.txt
None	53.51399302	119.4098661	50.50897098	123.9283819	41.17333102
component_id(btree)	36.13925004	122.898526	39.09370708	149.132715	26.45658684
component_id(btree), node_id(hash)	25.45816708	86.28342104	22.88536716	125.9463222	27.85415602
component_id(btree), node_id(btree)	38.65054893	117.9664488	32.99039006	155.06496	31.62352514
component_id(btree), node_id(hash), temp.node_id(hash)	40.20039201	109.426384	28.04028392	133.7669752	30.60440493
component_id(hash), node_id(hash)	26.92220187	90.23280811	24.43618298	210.5891101	29.8577292
component_id(btree), node_id(hash), link_table_name.dst_id(hash)	28.92120504	83.67425203	26.90488887	138.1120729	30.00807405

5.1.4 Eigenvalue computation (via Lanczos-SO and QR algorithms)

Index of columns in "WHERE" condition can help MySQL speed up value comparison in join operation. Therefore, we found that there are 7 possible positions to add index: G.row_id + G.col_id, Q.row_id + Q.col_id, R.row_id + R.col_id, Eval.row_id + Eval.col_id,

next_basis_vect.id, basis_vect_0.id, and basis_vect_1.id.
According to experiment result, we found that adding B-tree index on Eval.row_id and Eval.col_id improved the performance best. We also tried many index combinations, but the performance did not increase. Therefore, we decided to add B-tree index on index on Eval.row_id and Eval.col_id for Eigenvalue computation algorithm.

Index Type	Index column	p2p-Gnutella31	ca-AstroPh	email-EuAll	cit-HepTh	as-skitter.75000	Improvement
No cache		432.941635	42.247019	87.824524	34.531853	92.089166	(baseline)
Hash	G.row_id + G.col_id	183.792917	35.275701	76.680822	25.106503	83.686243	24.631%
Btree	G.row_id + G.col_id	212.340946	38.845005	95.304085	37.262694	137.772463	-1.405%
Hash	Q.row_id + Q.col_id	321.794434	28.335948	84.675914	37.241847	139.774482	0.511%
Btree	Q.row_id + Q.col_id	200.10483	28.930667	78.577032	34.081588	118.320216	13.729%
Hash	R.row_id + R.col_id	372.464409	41.754213	93.040661	34.649986	133.050895	-7.125%
Btree	R.row_id + R.col_id	282.987124	20.351836	56.000463	24.621752	72.129375	34.614%
Hash	Eval.row_id + Eval.col_id	373.817489	18.063141	60.053821	22.454597	80.1502	30.091%
Btree	Eval.row_id + Eval.col_id	66.592785	21.696901	58.072773	28.105395	99.974783	35.436%
Hash	next_basis_vect.id	683.642889	29.47404	87.384114	34.857184	81.874648	-3.404%
Btree	next_basis_vect.id	176.668195	22.145417	68.57327	38.863949	87.817504	24.157%
Hash	basis_vect_0.id	177.532078	41.937022	93.3785	39.510948	112.020925	3.468%
Btree	basis_vect_0.id	648.983299	20.541444	64.786556	25.261349	86.658346	12.090%
Hash	basis_vect_1.id	137.220548	21.072826	76.711587	26.999191	96.576688	29.603%
Btree	basis_vec_1.id	809.867548	26.501674	72.516903	27.561787	91.58051	-2.325%

5.1.5 Triangle count

This query is fully based on eigen value. And the aggregate function needs to go through all data. So index will not help.

Index	p2p-Gnutella31.txt	as-skitter.75000.txt	ca-AstroPh.txt	email-EuAll.txt	cit-HepTh.txt
None	0.000257969	0.000259876	0.000265121	0.00028801	0.000248194

5.1.6 K-core algorithm

Index of columns in "WHERE" condition can help MySQL speed up value comparison in join operation. Therefore, we found that there are 3 possible positions to add index: temp_degree_table.in_degree, temp_degree_table.node_id, and temp_link_table.src_id.
We tried all possible ways to add index on columns related to K-core algorithm; however, the experiment result showed that there was no significant improvement on executing time. Therefore, we decided not to add indices for K-core algorithm.

Index Type	Index column	p2p-Gnutella31	ca-AstroPh	email-EuAll	cit-HepTh	as-skitter.75000	Improvement
No cache		21.567541	35.087091	20.816424	80.608768	9.386785	(baseline)
Btreee	temp_degree_table.in_degree	30.652811	40.340461	23.426685	84.497394	6.604104	-8.963%
Hash	temp_degree_table.in_degree	28.243504	46.542064	32.58452	82.191118	13.88069	-33.994%
Btree	temp_degree_table.node_id	21.663131	37.245953	26.681098	103.65769	12.660117	-19.646%
Hash	temp_degree_table.node_id	21.885152	32.91817	25.688136	77.51908	6.447193	3.290%
Btree	temp_link_table.src_id	21.244998	46.501597	25.201246	86.955264	7.053383	-7.023%
Hash	temp_link_table.src_id	22.742343	40.598984	22.920802	90.440895	9.410922	-8.743%

5.1.7 Overall Validation

Next, we tested on several graphs about these indices. Some of them are sample graphs, some are new graphs for validating.

Most graph's processing time is shortened.
Notice that there is one graph's processing time becomes longer after we add indices. This might caused by the overhead for building the index on this graph.

With indices or not	ca-AstroPh	cit-HepTh	email-EuAll	p2p-Gnutella31	soc-Slashdot0811
Run time with no indices	2m34.348s	3m6.760s	8m34.530s	6m58.804s	6m33.274s
Run time with indices	1m50.844s	5m7.181s	7m27.525s	5m22.789s	4m7.349s

6 20 graphs Result

6.1 Experiment on 20 graphs

6.1.1 Degree distribution

Basically we use graphminer to get the degree distribution data. Then do loglog plot on the degree distribution.
The plots are shown in figure 1 to 20.
Based on these figures, we find some interesting observations.

Global Pattern
The figures of degree distributions show that most graphs follow power law.

Strange Behaviors

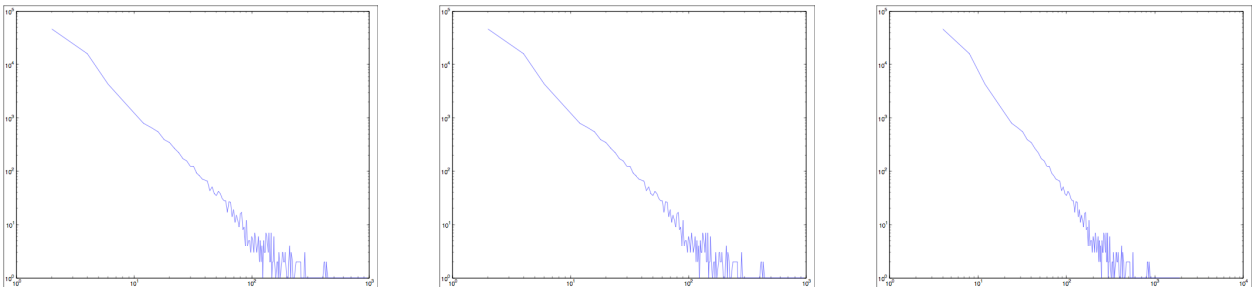
- a-AstroPh does not strictly follow power law. In figure 2 we can see that nodes with particular degrees are more than expect.

- There are spikes in p2p-Gnutella31 outdegree and degree distribution: figure 9. The reason for the spike may be because of default peer number of setting in p2p software.

- There is spike in email-Enron: figure 7. It might illustrates the most common connector numbers of a normal email user.

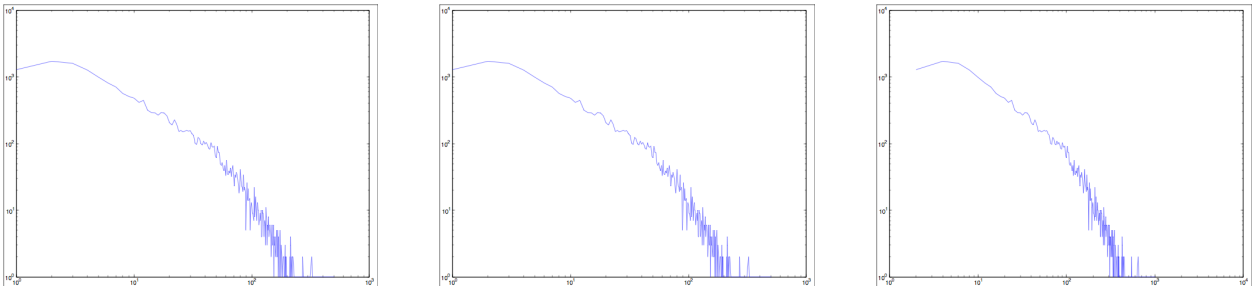
- There is spike in soc-Slashdot0811: figure 10. Similar to email-Enron, the spike might shows that a lot of users in Slashdot have about 2 friends.

- as-Caida does not strictly follow power law. In figure 11 we can see that numbers of nodes with degree 1 or 2 are very similar.



(a) In-Degree Distribution (b) Out-Degree Distribution (c) Degree Distribution

Figure 1: Degree Distributions of as-skitter



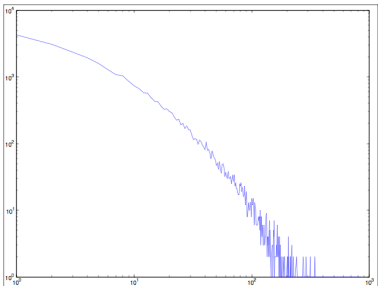
(a) In-Degree Distribution (b) Out-Degree Distribution (c) Degree Distribution

Figure 2: Degree Distributions of a-AstroPh

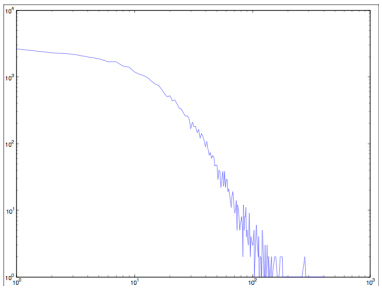
- There are lot of small spikes in soc-hamsterster: figure 16. They illustrate that there are many nodes in the graph have high degree. The graph tends to be highly connected.

- In soft-jdkdependency figure 19.There is a big group of nodes have similar out degree around 4 to 10. This indicates that jdk libs dependency are mostly within the scope of 4 to 10. And there is a small group of nodes which have very large out degree. These nodes should be very basic libs which are used by all other libs.

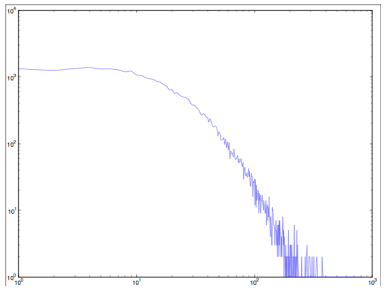
- There is a small spike in degree plot around degree 1 and 2 in text-spanishbook figure 20. These words could be idioms, which only followed by one or two specified words.



(a) In-Degree Distribution

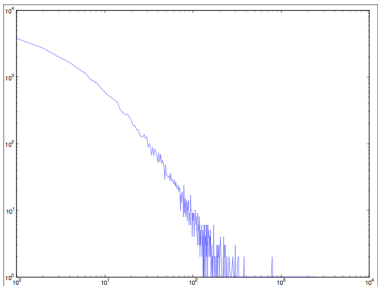


(b) Out-Degree Distribution

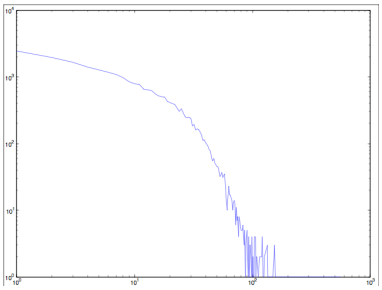


(c) Degree Distribution

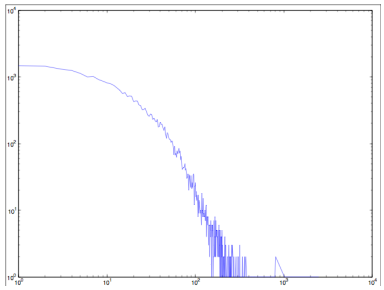
Figure 3: Degree Distributions of cit-HepPh



(a) In-Degree Distribution

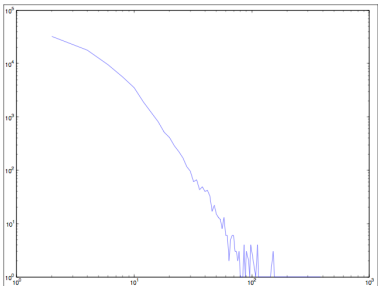


(b) Out-Degree Distribution

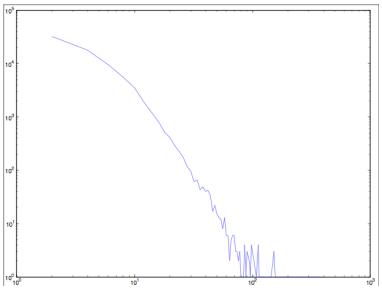


(c) Degree Distribution

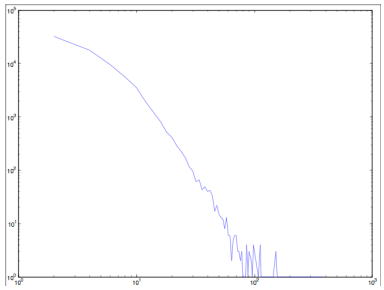
Figure 4: Degree Distributions of cit-HepTh



(a) In-Degree Distribution

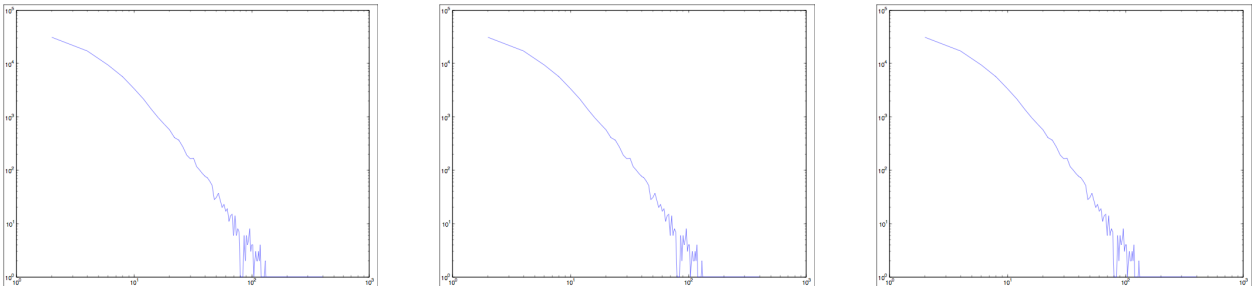


(b) Out-Degree Distribution



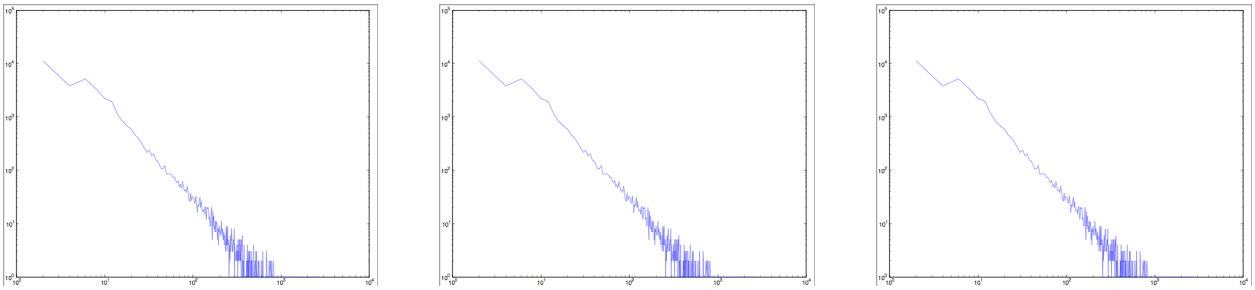
(c) Degree Distribution

Figure 5: Degree Distributions of com-amazon.ungraph



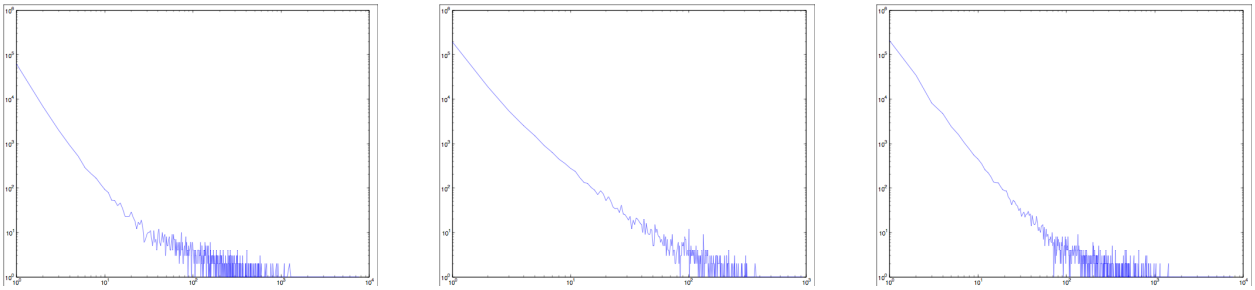
(a) In-Degree Distribution (b) Out-Degree Distribution (c) Degree Distribution

Figure 6: Degree Distributions of com-dblp.ungraph



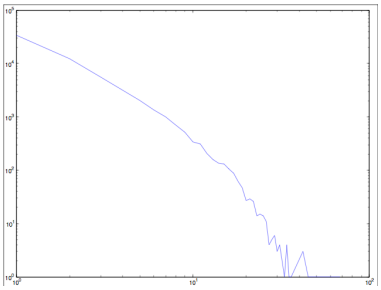
(a) In-Degree Distribution (b) Out-Degree Distribution (c) Degree Distribution

Figure 7: Degree Distributions of email-Enron.ungraph

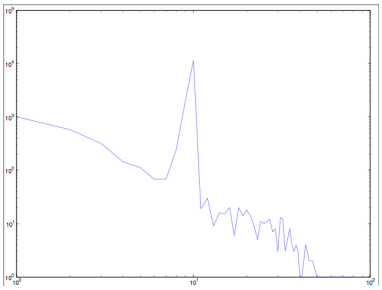


(a) In-Degree Distribution (b) Out-Degree Distribution (c) Degree Distribution

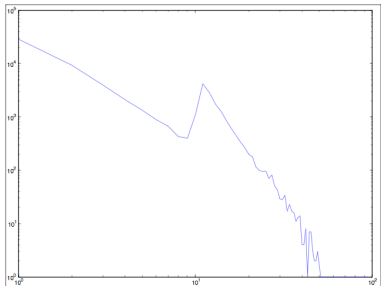
Figure 8: Degree Distributions of email-EuAll



(a) In-Degree Distribution

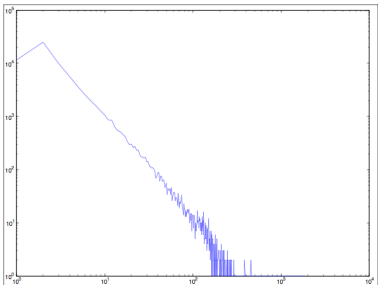


(b) Out-Degree Distribution

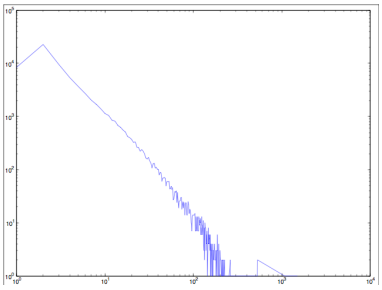


(c) Degree Distribution

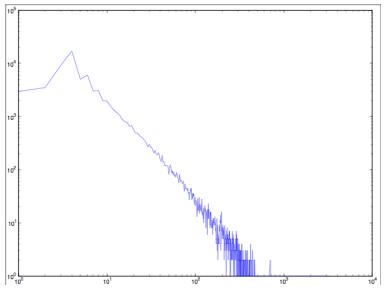
Figure 9: Degree Distributions of p2p-Gnutella31



(a) In-Degree Distribution

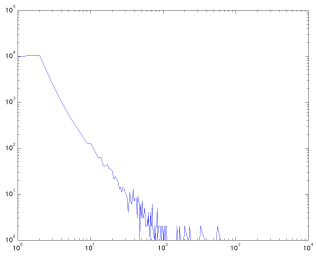


(b) Out-Degree Distribution

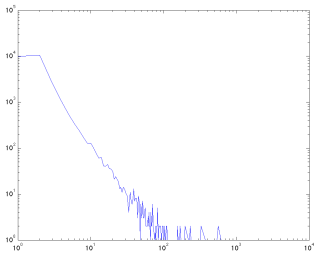


(c) Degree Distribution

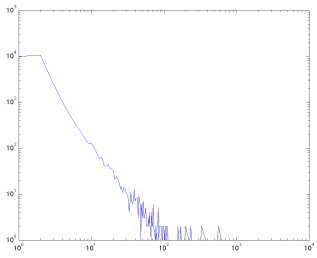
Figure 10: Degree Distributions of soc-Slashdot0811



(a) In-Degree Distribution

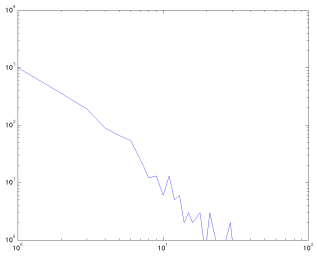


(b) Out-Degree Distribution

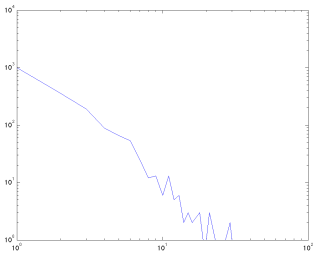


(c) Degree Distribution

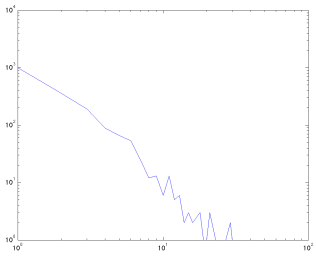
Figure 11: Degree Distributions of as-Caida



(a) In-Degree Distribution

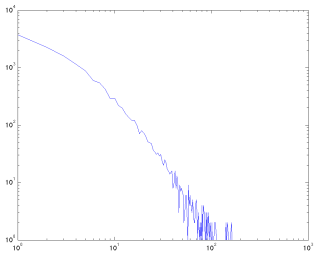


(b) Out-Degree Distribution

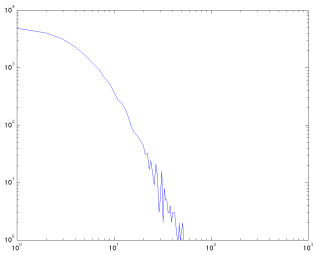


(c) Degree Distribution

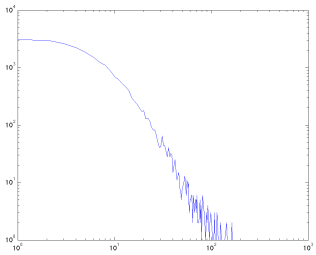
Figure 12: Degree Distributions of bio-protein



(a) In-Degree Distribution

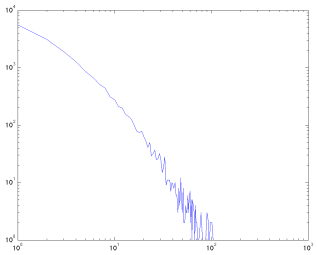


(b) Out-Degree Distribution

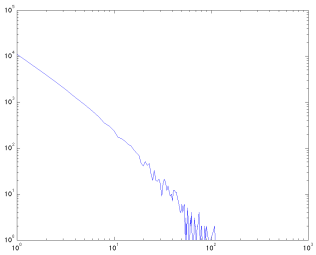


(c) Degree Distribution

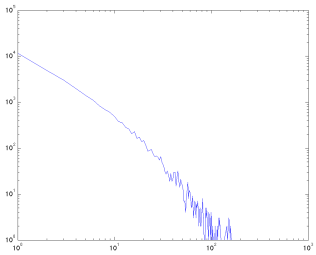
Figure 13: Degree Distributions of cit-Cora



(a) In-Degree Distribution

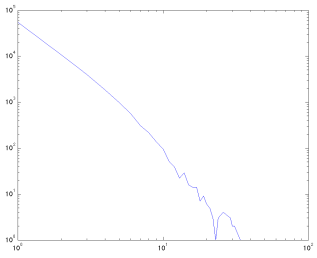


(b) Out-Degree Distribution

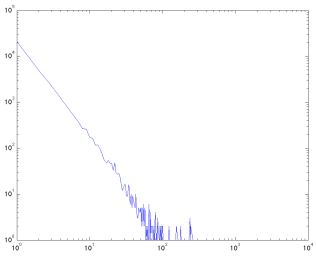


(c) Degree Distribution

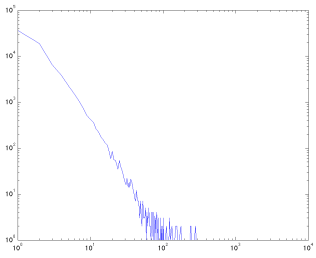
Figure 14: Degree Distributions of soc-digg



(a) In-Degree Distribution

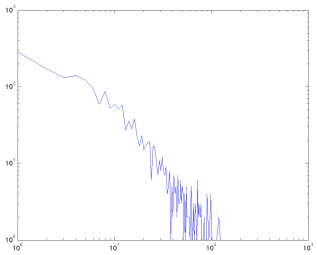


(b) Out-Degree Distribution

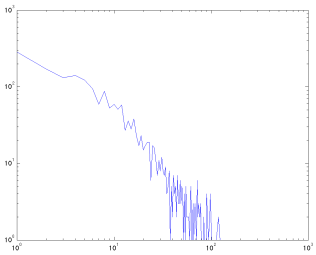


(c) Degree Distribution

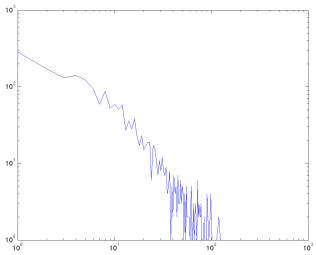
Figure 15: Degree Distributions of soc-flickr



(a) In-Degree Distribution

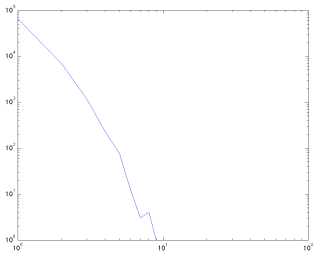


(b) Out-Degree Distribution

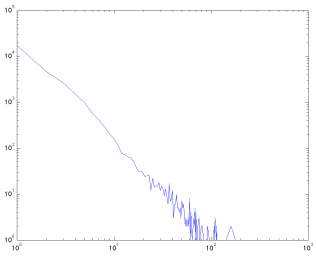


(c) Degree Distribution

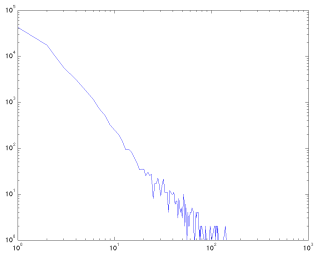
Figure 16: Degree Distributions of soc-hamsterster



(a) In-Degree Distribution

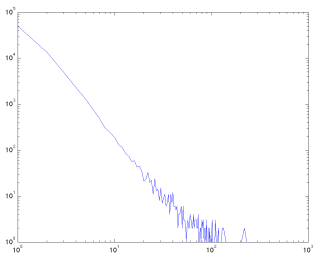


(b) Out-Degree Distribution

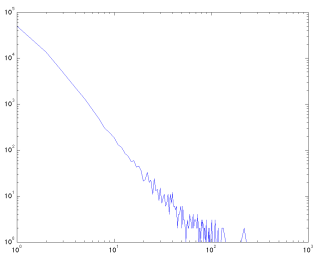


(c) Degree Distribution

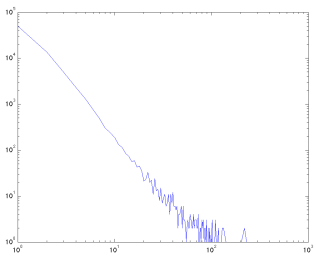
Figure 17: Degree Distributions of soc-pokec



(a) In-Degree Distribution

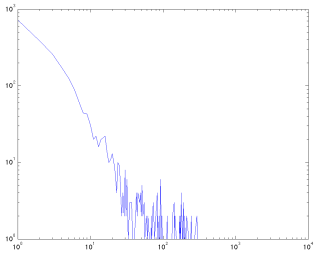


(b) Out-Degree Distribution

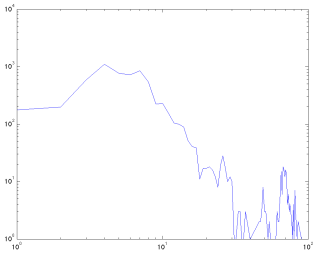


(c) Degree Distribution

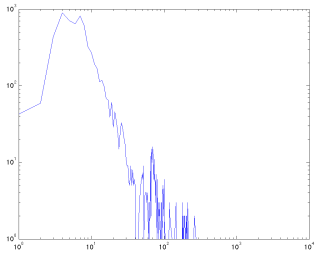
Figure 18: Degree Distributions of soc-Youtube



(a) In-Degree Distribution

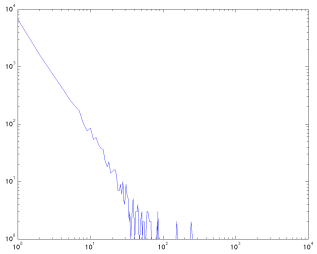


(b) Out-Degree Distribution

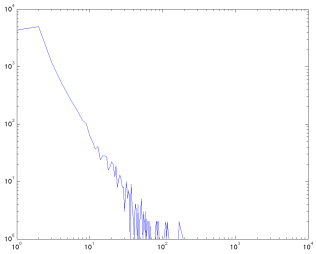


(c) Degree Distribution

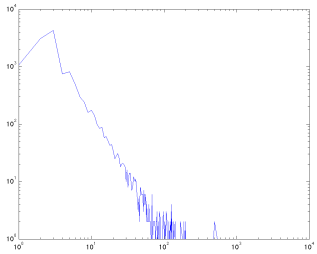
Figure 19: Degree Distributions of soft-jdkdependency



(a) In-Degree Distribution



(b) Out-Degree Distribution



(c) Degree Distribution

Figure 20: Degree Distributions of text-spanishbook

6.1.2 Weakly Connected Component and Triangle count

Firstly we use graphminer to get the connected component information and triangle count. For connected component data, we continue using matlab to get the components count and the number of nodes in each component. The result shows that most nodes are in a single component. So in this report, we also attached sizes of the biggest components in each graph.

Global Pattern

We find that most graphs consist of a small number of groups (compared with the total number of nodes). And most nodes are in one biggest groups.
And the triangle number illustrates how nodes in groups are connected to each other.

Strange Behaviors

- p2p-Gnutella31 has 12 components. This shows the nodes in this graph are highly connected. But the triangle count is little. So it seems that there are several popular nodes in this graph. And most nodes connect to these popular servers instead of connecting to each others.

- cit-HepPh has 61 components and the triangle count is a large number. This shows the nodes in this graph are mostly connected to each others.

- email-EuAll has 15836 components. Although the maximun component size is huge, the connections between nodes are not strong. It illustrates people are more likely to form small discussion groups in daily work communication.

- ca-AstroPh and email-Enron have extremely high triangle counts. Which shows the nodes in these 2 graphs are very densely connected.

- Nodes in as-Caida are all connected. And the triangles is much more than nodes number. Which indicates the graph is very dense.

- Nodes in cit-Cora are also all connected. But the triangle number is not large compared with the nodes number. So this graph is more likely to be a star type graph (highly centralize).

- In soc-hamsterster, triangles number is huge compared with members in each components. This shows that each components are highly connected. So the members in each components are very related with each other.

- soc-pokec has extremely small triangles number(4.8). Which indicates the graph is highly centralize in each sub groups.

- soft-jdkdependency and text-spanishbook both have large triangles numbers and all connected. The graphs are very dense.

Metrics	**as-skitter**	**ca-AstroPh**	**cit-HepPh**	**cit-HepTh**	**com-amazon**
components	310	290	61	143	1946
max group	69768	17926	34454	27465	47556
triangle	28389.34144	1061822.808	60696.51906	191035.2798	132.7590596

Metrics	**com-dblp**	**email-Enron**	**email-EuAll**	**p2p-Gnutella31**	**soc-Slashdot0811**
components	949	1065	15836	12	2091
max group	67361	33696	224832	62561	72780
triangle	786.338039	2059367.367	370075.0779	307.5803753	252186.8962

Metrics	as-Caida	bio-protein	cit-Cora	soc-digg	soc-flickr
components	1	173	1	373	1280
max group	26475	1458	23166	29652	63529
triangle	29967.0841542	30.1742701786	3192.44451312	5207.40905806	1574.17467495

Metrics	soc-hamsterster	soc-pokec	soc-Youtube	soft-jdkdependency	text-spanishbook
components	23	113	4319	1	1
max group	1788	73564	54143	6434	12643
triangle	15328.5846289	4.77180117934	100.71969417	150378.549887	191251.983102

6.1.3 PageRank

To analyze the patterns of PageRank in 20 graphs, we calculated the PageRank value of nodes in each graphs, aggregated the values into distribution, and plotted the distribution in log-log scale. Mainly, we wanted to check whether the distribution of PageRank values follow power law or not. At the same time, we observed the charts of distribution to find out strange patterns of PageRank.

Global Pattern

The experiment result (Figure. 21) shows that the PageRank value distribution in most of graphs follow power law.

Strange Behaviors

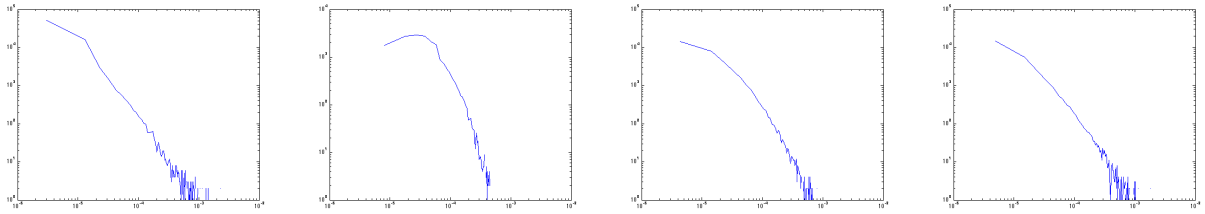
(1) Plateau in the Beginning

The distribution of PageRank in most of graphs follow power low; however, we found that there are plateaus in the beginning part of some distribution charts. Taking com-amazon.ungraph-75000 for example, the line between 10^{-8} and 10^{-5} is relative smooth.
ca-AstroPh, com-amazon.ungraph-75000, com-dblp.ungraph-75000, email-Enron.ungraph, soc-Slashdot0811-75000, as-Caida, soc-flickr-75000, soc-pokec-75000, soc-Youtube-75000, and text-spanishbook have such plateau pattern.

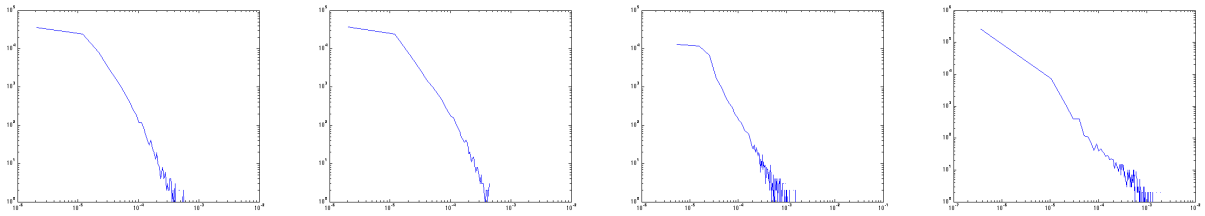
(2) Broom Tail

Ideally, the chart of distribution following power law would be a smooth descending straight line in log-log scale plot; however, we found that some lines of real data are not as smooth as ideal straight lines. Instead, the lines of some real data jump up and down in the last part. The shape is like a bloom.
as-skitter, cit-HepTh, email-Enron.ungraph, email-EuAll, as-Caida, cit-Cora, soc-Youtube-75000, soft-jdkdependency, and text-spanishbook have such broom tail pattern.

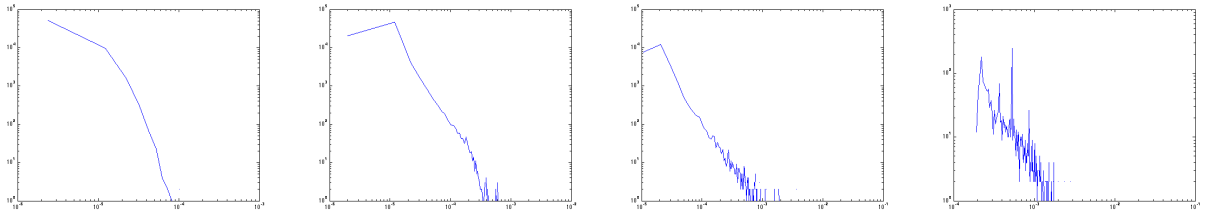
(3) Vibrating Line



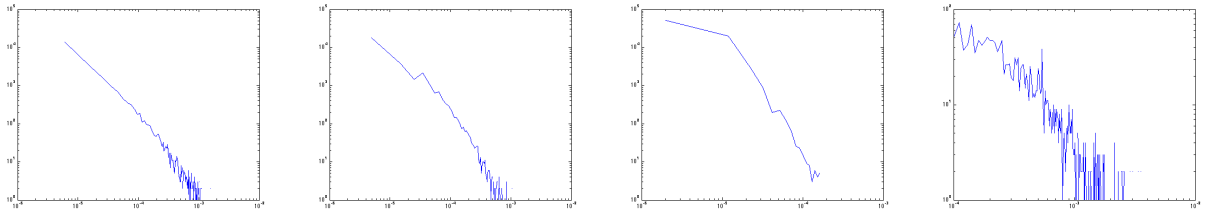
(a) as-skitter.75000 (b) ca-AstroPh (c) cit-HepPh (d) cit-HepTh



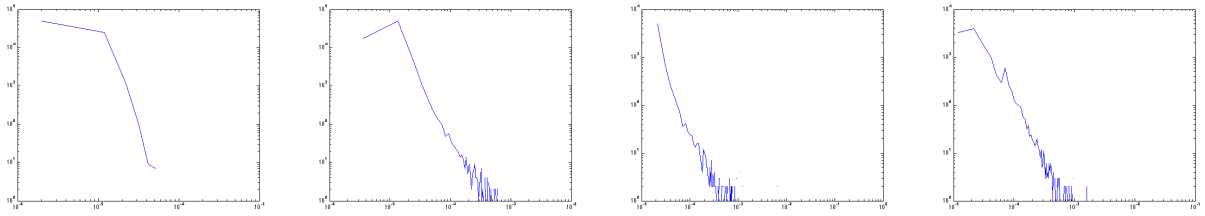
(e) com-amazon.ungraph-75000 (f) com-dblp.ungraph-75000 (g) email-Enron.ungraph (h) email-EuAll



(i) p2p-Gnutella31 (j) soc-Slashdot0811-75000 (k) as-Caida (l) bio-protein



(m) cit-Cora (n) soc-digg (o) soc-flickr-75000 (p) soc-hamsterster



(q) soc-pokec-75000 (r) soc-Youtube-75000 (s) soft-jdkdependency (t) text-spanishbook

Figure 21: Pagerank value Distributions of 20 graphs

The lines in PageRank distribution chart are usually smooth, but there are some lines vibrating severely. Taking soc-hamsterster for example, its line jumps up and down severely, although the trend of line follow power-law. bio-protein and soc-hamsterster have such pattern.

6.1.4 Eigenvalue computation (via Lanczos-SO and QR algorithms)

	cit-HepPh	**com-amazon.ungraph-75000**	**com-dblp.ungraph-75000**	**email-Enron.ungraph**	**soc-Slashdot0811-75000**
First Eigenvalue	71.24869215	12.46375985	18.08047143	232.0718265	124.3376448
Second Eigenvalue	15.78100055	-10.49450063	-9.989018866	-52.74314639	-74.23852603
Third Eigenvalue	-11.28234916	2.528983265	3.950839681	16.07981363	3.277331459

	p2p-Gnutella31	**ca-AstroPh**	**email-EuAll**	**cit-HepTh**	**as-skitter.75000**
First Eigenvalue	12.26069439	182.7509283	134.5445114	108.0148181	182.7509283
Second Eigenvalue	2.714800955	64.42806942	-59.94291686	-48.59746457	64.42806942
Third Eigenvalue	-2.60170164	-0.449525922	6.537952019	9.103275079	-0.449525922

	as-Caida.undir	**bio-protein-undir**	**cit-Cora**	**soc-digg**	**soc-flickr-75000**
First Eigenvalue	68.3693197062366	6.59050522092269	27.0826721488691	31.4948560594139	46.128302835195
Second Eigenvalue	-51.89780364579	-4.73938936859157	-9.39990749609768	3.5437520638839	-44.5991249417485
Third Eigenvalue	0.336124325308012	1.07545809389524	4.9442935842295	-3.43736182466055	1.53574108698456

	soc-hamsterster.undir	**soc-pokec-75000**	**soc-Youtube-75000.undir**	**soft-jdkdependency**	**text-spanishbook**
First Eigenvalue	45.2761211252026	9.38962714926706	15.7609882639711	143.265820052228	124.70922072648
Second Eigenvalue	-10.0662421612482	-9.28310856022425	-14.9067351906316	-126.79096036676	-92.5441274101652
Third Eigenvalue	5.6332490229777	0.918759752366816	1.16658315444281	2.26001659190722	8.30033058289115

6.1.5 K-core algorithm

To analyze K-core value, we set k = 5, applied K-core algorithm on 20 graphs, and counted the distribution of k-core size. We were curious about the patterns of the K-core size distribution.

Global Pattern

According to the result, we found that the graphs can be grouped into two types of graphs: Type I graphs and Type II graphs.

Type I Graphs

Type I graphs, like soc-Slashdot0811-75000, p2p-Gnutella31, email-EuAll, email-Enron.ungraph, cit-HepTh, cit-HepPh, ca-AstroPh , as-Caida, cit-Cora, soc-digg, soc-flickr-75000, soc-hamsterster, soft-jdkdependency, and ca-AstroPh, have a giant 5-core with more than 1000 nodes, which means that type I graph is densely connected.

Type II Graphs

Type II graphs, like as-skitter.75000, com-dblp.ungraph-75000, com-amazon.ungraph-75000,

bio-protein, soc-pokec, and soc-Youtube-75000, do not have such a giant k-core. Instead, there are many small (size < 500) 5-cores in type II graphs. Moreover, some graphs even do not have 5-cores, like bio-protein, soc-pokec, and soc-Youtube-75000. It indicates that Type II graphs are not densely connected.

soc-Slashdot0811-75000		p2p-Gnutella31		email-EuAll		email-Enron.ungraph		cit-HepTh		cit-HepPh		ca-AstroPh	
size	frequency	size	frequency	size	frequency	size	frequency	size	frequency	size	frequency	size	frequency
26137	1	16174	1	7030	1	6	6	21181	1	28593	1	6	2
						7	2					7	2
						8	3					8	1
						9	1					11	1
						12	1					18	1
						15	1					12236	1
						11538	1						

as-Caida		cit-Cora		soc-digg		soc-flickr-75000		soc-hamsterster		soft-jdkdependency		ca-AstroPh	
size	frequency	size	frequency	size	frequency	size	frequency	size	frequency	size	frequency	size	frequency
1192	1	9355	1	6794	1	1122	1	1070	1	4869	1	3144	1

5-core distribution in Type I graphs

as-skitter.75000		com-dblp.ungraph-75000		com-amazon.ungraph-75000		bio-protein		soc-pokec		soc-Youtube-75000	
size	frequency	size	frequency	size	frequency	size	frequency	size	frequency	size	frequency
14	1	6	9	6	1	6	1	0	0	0	0
181	1	7	3								
		8	3								
		9	1								
		10	1								
		13	1								
		15	1								

5-core distribution in Type II graphs

7 Division of Labour

- Jiajung Wang:
  - Degree Distribution analysis
  - Weakly Connected Component analysis
  - Triangle count analysis
  - Organize latex files
- San-Chuan Hung:
  - PageRank analysis
  - K-core analysis
  - Eigenvalue analysis
  - Organize latex files

8 Conclusion

We implemented K-core algorithm into GraphMiner, and used indexing technology to speed up execution time. According to the experiments, the indexing method can reduce computing time.

Besides, we used revised GraphMiner to analyze 20 graphs in different aspects. We found global patterns across graphs, and some special patterns in some graphs.

Last, we provided friendly scripts to analyze graph and to do unit test. These scripts can be found in graphminer directory.

References

[1] José Ignacio Alvarez-Hamelin, Luca Dall'Asta, Alain Barrat, and Alessandro Vespignani. k-core decomposition: a tool for the visualization of large scale networks. *arXiv.org*, April 2005.

[2] U Kang, Brendan Meeder, and Christos Faloutsos. Spectral analysis for billion-scale graphs: discoveries and implementation. In *PAKDD'11: Proceedings of the 15th Pacific-Asia conference on Advances in knowledge discovery and data mining*. Springer-Verlag, May 2011.

[3] U Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. gbase: an efficient analysis platform for large graphs. *The VLDB Journal — The International Journal on Very Large Data Bases*, 21(5), October 2012.

[4] U Kang, Charalampos E Tsourakakis, Ana Paula Appel, Christos Faloutsos, and Jure Leskovec. HADI: Mining Radii of Large Graphs. *Transactions on Knowledge Discovery from Data (TKDD*, 5(2), February 2011.

[5] U Kang, Charalampos E Tsourakakis, and Christos Faloutsos. Pegasus: mining peta-scale graphs. *Knowledge and information systems*, 27(2):303–325, 2011.

[6] 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'11: Proceedings of the 2011 European conference on Machine learning and knowledge discovery in databases*. Springer-Verlag, September 2011.