Subgraph Frequencies and Network Classification

Quaizar Vohra

1. Introduction

Current metrics used in summarizing networks, such as degree distribution, average diameter and clustering coefficient, provide a very coarse understanding of their structure. One would like to have finergrained summaries of networks which allow making distinction between structurally different networks. A class of such interesting graph statistics are the Small Subgraph Frequencies. These subgraph frequencies allow us to study the underlying structure of networks by distinguishing mathematical properties from social behavior related properties.

Given a small graph H (template) of t nodes and a large graph (network) G, the subgraph frequency $\#(H,G)$ is the fraction of t-tuples of nodes of G such that the induced graph is H. One can consider the subgraph frequencies for all possible graphs H on t nodes and summarize the network by stacking their frequencies into a vector. This method was proposed in [1] and was studied as a local property for a large collection of small subgraphs in a large social network. The results from this study suggest that studying the frequency statistics of large networks as a single unit may provide unique insights. To our knowledge, this statistic has not been studied as a global property of a large network.

We first implement advanced methods for computing subgraph frequencies for very large networks. This allows us computing subgraph frequency statistics for a large number of artificial networks and real world networks found in the SNAP database. This statistic will allow us to classify these networks into different groups (using a clustering mechanism) giving us the mapping from real-world networks to their closest aritificial models. More details on our implementation is presented in section 3 and 6. The study study of subgraph frequency statistic on real networks is presenented in section 8.

We implemented the scheme described in [2], which uses an unbiased estimator which approximates the occurrence of H in G. The error is reduced by running a very large number of parallel instances of this estimator. Section 3 and 4 describes this implementation and subsequent results, respectively. The results were not very encouraging as the estimations made by this scheme are not accurate inspite of using a very large number of estimators. The later part makes this scheme very inefficient as well. We explain the possible causes for this inaccuracy in the same section.

This led us to explore another scheme for subgraph counting which is described in [11]. This is a sampling algorithm which approximates all 4-node subgraph frequencies. This scheme is very efficient and gives us accurate results as well. Its implementation and results are described in section 5 & 6 respectively.

Section 7 describes the use of subgraph frequency as a metric in studying network properties. Specifically we fit Synthetic Model Graphs of different types to Real Networks and then use this metric along with other graph metrics to compare the fitted model with the real network. We do this experiment on the SNAP DB with results described in section 8.

2. Prior Work

This section describes the prior work relevant to our project in more details. The first paper describes why subgraph frequency statistic is interesting. The last 3 describe various schemes for counting subgraphs in a large graph.

Use of Subgraph Frequency Statistic in Studying Network Properties

The paper by Ugander et. al [1] demonstrates the use of subgraph frequencies in studying network properties. They found that pure mathematical constraints limit the occurrence of certain structures (subgraphs) while social behavior prevent creation of other structures, e.g a triangle with an edge missing. Their main technique is to create a coordinate system based on a vector of subgraph frequencies where each coordinate encodes the relative frequency of a distinct k-node subgraph H in a larger graph G. Typical value of H is small, i.e. 3 or 4 nodes. For example, for $V(H) = 3$ there are only 4 distinct subgraph types and hence the vector is 4-dimensional.

They demonstrate the usefulness of subgraph frequencies as features for classification of networks by accurately classifying 3 types of networks on facebook, e.g. friendship neighborhoods, facebook groups and event networks. They concluded that networks with different structures (represented by different subgraph frequency vectors) point to different human behavior when faced with different settings.

The measurements done in this paper use a large collection of small dense graphs (50 to 200 nodes each) induced from the real facebook networks. One Subgraph frequency vector is computed per induced network. The results of this paper consistently show that subgraph frequencies of real world networks lie very close to a 1-dimensional band and this band becomes narrower as the size of the network grows.

Counting Arbitrary Subgraphs in Data Streams

This is the scheme we have implemented and is described in detail in section 3.

Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts

This is the second scheme we implemented for counting SubGraphs and is described in detail below.

3. Counting Subgraphs Algorithm Implementation

Glossary
$[n]$: the arithmic field $[0, 1, 2, ..., n-1]$
G : input graph in which subgraph frequency is being measured
m : # of edges in G, i.e. $|V(G)|$
H : subgraph or template graph being searched in G. For matching H with subgraphs in G, we assume an arbitrary orientation of each edge in H
k : # of edges in H, i.e. $|V(E)|$
t : # of vertics in H, i.e. $|V(H)|$
T : edge induced subgraph in G with k edges. Each edge is assumed to have a specific orientation (direction) for matching with H
a, b, c : nodes in H for description purposes u, v, w : nodes in T or G for description purposes $deg_G(v)$: degree of node v in graph G
$X_c(w)$: A $4k$-wise independent hash function which assigns each node w in G with a random $deg(c)$-th root of unity
Q : A random τ-th root of unity chosen independent of each X_c where $\tau = 2^t - 1$

$Y(w)$: A $4k$-wise independent hash function which assigns each node w in G with a value in $S = \{2^0, 2^1, ..., 2^{t-1}\}$. This is again chosen independent of X_c and Q
$\theta_T(c, w)$: amount of degree of node c in H contributed by a node w in T (when comparing T with H). This is the number of edges in T with head or tail w which match an edge in H with c at its head or tail, respectively. Since the algorithm does an edge-by-edge comparison of H & T, for T to be homomorphic to H, there has to be exactly one node w in T which accounts for all the edges of node c in H (whether at the head or tail) and hence its degree (note that same w might contribute to all the degree of multiple nodes in H)

Algorithm

This algorithm is based on estimating the number of edge-induced subgraphs T of G which are isompor-phic to H. This estimation is done using a randomized algorithm which indirectly considers each edge-induced subgraph of G, called T. T has a specific edge orientation for comparision with H. This algorithm has 2 parts: the first part models whether T is homorphic to H and given that it then models whether T is an isomporphism of H. These part are described in turn below.

Modelling Homomorphism: T is homomorphic to H, if for every node c in H there is exactly one node in T which contributes all the degree of c in H. This is modeled by $X_c(w)^{\theta_T(c,w)}$ which is one only if $deg_H(c) = deg_T(w)$ because $X_c(w)$ is the $deg_H(c)$-th root of unity. Entire T is modelled by the product of all $X_c(w)$ terms (for all nodes c in H and all nodes w in T), i.e. $\prod_{c \in V(H)} prod_{w in V(T)} X_c(w)^{\theta(c,w)}$ which is one in expectation only when T is homomorphic and 0 otherwise. This is because $X_c(w)$ will be given different $deg(c)$-th roots of unity across different instances of the estimator and it will be zero in expectation unless it is raised to $deg(c)$ which only happens if only one node w in T contributes towards all of the degree of node c in H. Note that here there is a strong dependence on the assumptions that $X_c(w)$ are $4k$-wise independent (i.e. $X_c(w)$ is independent of $X_c(x)$ and also of $X_b(y)$ for the expectation to work across estimators so that expectation of product can be converted into product of expectation).

Modeling Isomoprhism: Given T is homomorphic to H, it is isomporhic if every vertex in H is matched by a vertex in T. This is modelled by the $\prod_{c \in V(H)} prod_{w in V(T)} Q^{\frac{\theta_T(c,w) Y(w)}{deg_H(c)}}$. Here again every matching of node w in T to c in H (via the head/tail comparision of edges in H and T) contributes $Q^{\frac{Y(w)}{deg_H(c)}}$ factor in the product. Given T is homomorphic to H (driven by the product of X_C variables as described above), for every node c in H, there is exactly one node w in T such that $\theta_T(c, w) = deg_H(c)$ and so the product above reduces to $prod_{w in V(T)} Q^{\sum_w Y(w)}$. This product is non-zero (a constant equal to $\frac{t!}{t^t}$, see [1] for the exact computation) in expectation (across estimators) only if $\sum_w Y(w) = 2^t - 1$ as Q is the $(2^t - 1)$-th root of unity. This is true only if $Y(w)$ for all the nodes in T cover the entire set $S = \{2^0, 2^1, ..., 2^{t-1}\}$ (see [2] for proof). Since $|S| = t$, there are exactly as many nodes in T as in H, hence T and H are isomorphic.

The 2 products above, $\prod_{c \in V(H)} prod_{w \in V(T)} X_c(w)^{\theta(c,w)}$ and $\prod_{c \in V(H)} prod_{w \in V(T)} Q^{\frac{\theta_T(c,w) Y(w)}{deg_H(c)}}$, together model the isomporphism of H to T. Their result is zero in expectation if T is not isomprophic to H and a constant otherwise.

The above computation was shown in detail to give the intuition behind the estimator and highlight the role of expectation and independence of the random variables. It is not performed in reality as the number of T's in G is $O(m^K)$ which is very large. The actual computation is done by simply considering one edge in G at a time and calculating $X_a(u) X_b(v) Q^{\frac{Y(u)}{deg_H(a)}} Q^{\frac{Y(v)}{deg_H(b)}}$ for both orientations of (u,v) (i.e. (u,v) as well as (v, u)) and adding them to a running esimator $Z_{(a,b)}(G)$ for every edge (a, b) in H. Taking the product $\prod_{(a,b) \in H} Z_{(a,b)}(G)$ indirectly gives us the above 2 products for every T in G. This clever trick gives us a very efficient though probabilistic estimation of $\#(H, G)$.

The number of estimators required is computed by applying Chebyshev's inequality on the variance of

$Z_{(a,b)}(w)$. The paper shows that it takes $O(\frac{m^K(\Delta G)^K}{\epsilon^2 \cdot \#(H,G)})$ estimators to acheive an error bound of (1) with probability greater than 2/3.

Note the heavy dependence on the assumption of indepdenence across all the random variables (including the ones which are generated using X_c hash functions). In the worst case, T has $2k$ nodes (either ends of k nodes) and hence the expectation of the product requires $2k$-wise independence so that one can convert the expectation of product into product of expectations. The use of variance in determining the count of estimator further tightens this requriement to $4k$-wise independence.

Pseudocode

```
class $Estimator$:
  def $constructor(G, H)$:
     t $\leftarrow NodeCount(H)$
     $\tau \leftarrow 2^t - 1$
     $Q \leftarrow$ random $\tau^{th}$ root of unity
     for each c in H:
       $X_c \leftarrow$ random 4k-wise hash fn with domain as $[deg(c)]$ and range as $[t]$
     $Y \leftarrow$ random 4k-wise hash fn with domain as $[|V(G)|]$ and range as $[t]$
     for each $(a,b) \in E(H)$:
       $Z_{(a,b)} \leftarrow 0$
     $XQprod \leftarrow \{\}$
     for each $(w, c), w \in V(G), c \in V(H)$:
       $XQProd[(w, c)] \leftarrow X_c(w) \cdot Q^{\frac{Y(w)}{deg(c)}}$

  def $updateZ((u,v))$:
     for each $(a,b) \in V(H)$:
       $Z_{(a,b)} \leftarrow Z_{(a,b)} + XQProd[(u, a)] \cdot XQProd[(v, b)] + XQProd[(u, b)] \cdot XQProd[(v, a)]$

  def $estimate()$:
    for each $(u, v) \in E(G)$:
      $est.updateZ((u,v))$

    return $\prod_{(a,b)\in H} Z_{(a,b)} \cdot \frac{t^t}{auto(H) \cdot t!}$          // auto(H): # of automorphisms of H
// End of Estimator Class


def $countSubgraphs(G, H)$:
  $numEstimators \leftarrow$ compute # of of estimators based on tolerance $\varepsilon$, $|E(G)|$ & expected $\#(H, G)$
  $total \leftarrow 0$
for each $i \leftarrow 1$ to $numEstimators$:
   $est \leftarrow Estimator(G, H)$
   $total \leftarrow total + est.estimate()$
return $total/numEstimators$
```

The first part in the pseudocode describes the *Estimator* class which allows us to create several instances of an estimator. The constructor of the estimator initializes the hash functions, X_c for each node c in H and Y; and the random variable Q. It precomputes the product of $X_c(w)$ and $Q^{\frac{Y(w)}{deg(c)}}$ for every pair of (w, c) for every node w in G and every node c in H. The reason for this precomputation is to make our main loop of updating our estimator Z for every edge (u, v) in G efficient. The assumption here is that the number of edges edges far exceeds the number of of nodes in a graph.

The method *updateZ*, takes one edge (u, v) of G and updates $Z_{(a,b)}$ for every edge (a, b) in H. The method *estimate* iterates thru all the edges of G and calls the *updateZ* method on that edge. After iterating through all the edges, it returns the final estimate.

Function *coutingSubgraphs*() implements the counting algorithm by first computing the number of estimators required based on the desired tolerance ϵ, number of edges m in G and the expected number of occurances of H in G based on edge density of G (i.e. the probability of an edge occuring between any pair of nodes in G)

3.1. K-wise Hash Functions

While implementing this algorithm we quickly realized the importance of having $4k$-wise independent hash functions. This led us into the exploration of $4k$-wise hash functions, i.e what is their definition and how can one construct those.

Specifically, here is a list of issues we have to address regarding the application of k-wise independent hash functions to our context.

- How to construct a k-wise independent hash function: there is a simple polynomial based construction described in [5] as follow:

$$\sum_{i=\{1,2,\ldots,k\}} a_i x^{i-1} \bmod p$$

  where p is prime and a_i's are randomly and independently chosen coefficients in $[p]$. Here the assumption is that both the domain and range of the hash function is $[p]$

- How to construct k-wise independent hash function whose range is smaller than its domain: As per the current state-of-art, it seems this is possible only if the range and domain are both powers of the same number. This problem was addressed in [6]. In absence of this condition, there is no perfect k-wise independence. One could use the polynomial construction as above and then take $module(|range|)$ to get approximate k-wise independence.

- How to efficiently constuct k-wise independent hash functions: this is described in [7] which is known as tabulation based scheme - the idea here is to break a long hash key into q characters (a char could be 8 or 16 bits or some fixed small # of bits). Generate $(k-2)(q-1)$ derived chars if k-wise independence is desired. Then hash the resulting $q + (k-2)(q-1) = (k-1)(q-1) + 1$ chars using as many hash functions (chosen randomly and independently from k-wise hash family) and then taking an XOR of their results. Note that the domain of these hash functions is a character and hence can be tabulated. The polynomial based hash construction as described above is used for computing these character-wise hash functions

- How does one pick a set of k-wise hash functions independently across many estimators of our subgraph counting algorithm: we just use the python random number generator to get the polynomial coefficients in the polynomial based constuction of hash functions

We could not find any publically available implementation of k-wise hash functions. This led us to develop our own. This code resides at [8].

4. Counting SubGraphs: Experiments & Results

Setup

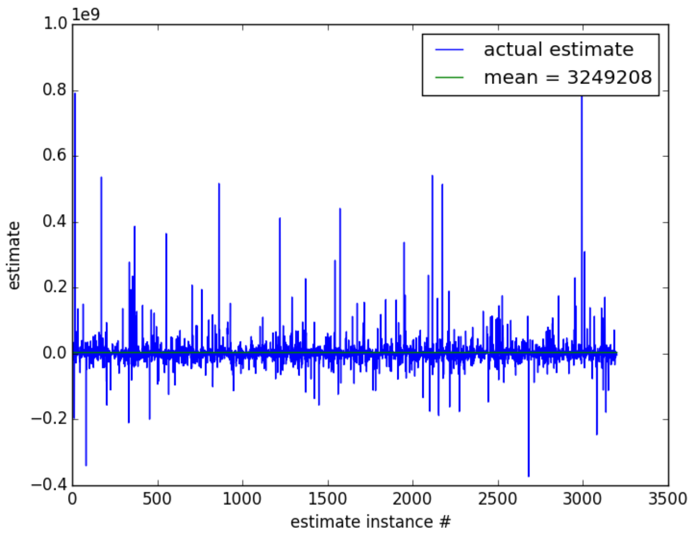
The first thing we wanted to ensure was that our implementation of [2] gives accurate estimates before worrying about scaling tests. To check this, we create a Erdos-Renyi random graph with 500 nodes and 60000 edges.

We kept the edge density $p = 1/2$ so that the number of estimators required is reasonably small. We experimented with counting triangles in this graph.

As explained in section 3, the number of estimators required is $O(\frac{m^K (\Delta G)^K}{\epsilon^2 \cdot \#(H,G)})$. For a graph which is a cycle with n nodes, m edges and edge-density as p, one can simplify this to $O(\frac{1}{\epsilon^2 p^k})$. For a triangle, this number scales inversely proportional to the cube of p and square of ϵ. Assuming $p = 1/2$ and $\epsilon = 0.05$, we require 3200 instances of estimators.

Results

The following plot shows the variance in the count estimates across the 3200 estimator instances. The x-axis tracks the estimator instance number while the Y axis tracks the subgraph count estimated by each estimator. Notice the huge variance around the mean.



The results from this experiment leads us to conclude that estimator is not very accurate. The estimated count vary between ±33%. Our graph contained 2587236 triangle while our estimated count vary between 1990000 and 3250000. Currently this experiment take 30 minutes to run (using 3200 estimator instances) despite the precomputations we do to speed our main estimator loop (described in section 3).

To analyze the variance of our estimator, we ran the above experiment on a smaller graph with 100 nodes (keeping the edge-density p same). This allowed us to run our algorithm a large number of times and compute the variance. This graph has 20136 triangles. The mean and standard deviation from 100 runs of our algorithm is 22933 and 6576, respectively.

Why are the results not accurate

Here are some of the reasons which may have caused the inaccuracies in our results:

- Perfect k-wise independence lacking due to the domain and range of our hash function not being a power of the same number.
- We are using the python random generator to pick the polynomial coeffecients which may not be yielding sufficiently independent hash functions
- We are again using python random generator to pick Q value which may not be sufficiently independent of the hash functions, X_c and Y

Note on performance

Note that in the above experiment we were careful to keep the edge-density p high so that the number of estimator instances required is reasonable. If the graph was sparse, we would need far more estimators. For example, we considered the facebook ego network in the SNAP database which has 4000 nodes and 7000 edges and $p = 1/200$. This would require 3.2 billion instances of estimators for $\epsilon = 0.05$ as the algorithm scales inversely proportional to p^3). Given that it takes 30 minutes to run 3200 instances of the estimator on a dense graph with 60000 edges, the time it will take to complete this algorithm with 3 billion instance would be very long. So we could not perform this experiment on real-world networks.

This algorithm is unusable for sparse graphs. This algorithm works well for only counting paths and stars in real-world networks. The algorithm is highly parallelizable so we could run in a compute cluster.

5. Path Sampling Algorithm & Implementation

This scheme is described in [11]. It is a sampling algorithm which counts frequencies of all 4-vertex subgraphs. It is efficient and accurate (requires only 200K samples for 1-2% error on graphs of the order of 100M edges) with provable error bounds. The scheme uses 3-path sampling (a 3-path is a connected 4-node subgraph with 3 edges and is not a star) as a building block around which counting of all other patterns is built. It further describes a pruning scheme to decrease the variance in estimates for the case of 4-edge cycle, 5 and 6-edge subgraph cases.

The 3-path sampling is very simple and elegant. They first show that there is a linear relationship between induced 4-node subgraph and a non-induced 4-node subgraph. This allows one to accurately compute the probablity of sampling a non-induced 3-path subgraph contained in a particular induced 4-node subgraph pattern (e.g. every induced 4-node cycle contains exactly 3 distinct 3-path non-induced subgraphs etc.). Next they assign each edge $e = (u, v)$ a probability proportional to $\tau_e = (d_u - 1)(d_v - 1)$ where d_u is the degree of of node u. This makes the probability of selecting a 3-path with (u, v) as the middle edge (i.e. $\{(u', u), (u, v), (v, v')\}$) a constant.

Edges (u, v) are randomly sampled using probability proportional to τ_e and neighbors of u and v are randomly selected to give a 3-path, $\{(u', u), (u, v), (v, v')\}$. Then the 4-node subgraph pattern induced upon this 3-path is counted. By repeatedly sampling a large number of 3-paths and using the linear relationship of a non-induced subpath being found in a induced subgraph, it is easy to estimate the expected value of each induced subgraph pattern. They also prove some linear bounds on error and confidence of these estimates using Hoeffding inequality.

They show that the above algorithm is not accurate enough for 4-edge cycle and 5 and 6-edge motifs. They next refine this algorithm by restricting the 3-paths sampled to 4-edge cycles. This reduces the the sample space significantly improving the accuracy for the larger motifs. Please read [11] for more details

6. Subgraph Frequencies as a Graph Metric

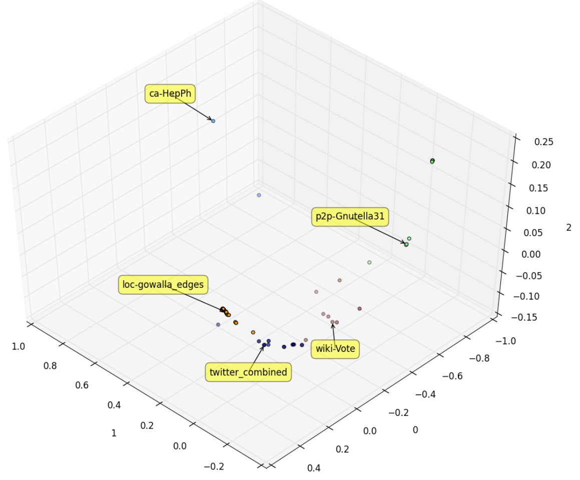
A network can be characterized by a vector of subgraph frequencies as demonstarated in [1]. Using the scheme in section 5, we compute 4-node subraph frequencies for all subgraphs of a particular size (number of nodes) and stack those frequencies into a vector. The vector maps a graph to a point in a multi-dimensional space using this vector. This scheme is very similar to [1]. The only difference is that we do not take into account frequencies of disconnected subgraphs.

In particular, we compute subgraph frequencies for all 4-node subgraph patterns for all networks in the SNAP DB which are stacked up into subgraph frequency vectors. We then use PCA to get a low dimensional project of these vectors. Next we run k-means to categorize these networks into groups. For each group, we select a representative which is closest to the centroid of the cluster.

7. Subgraph Frequencies: Experiments & Results

Using PCA, we found that most of the variance of the subgraph frequency vector can be explained by a 3 dimensional project. The percentage of variance retained by these 3 dimensions are $[81.6\%, 14.4\%, 3.6\%]$. Thus these 3 dimensions retain 99.6% of the variance in the subgraph frequency data. In fact the first 2 dimensions explain most of the variance.

The results of running k-means are shown in the following 3-D plot. This plot also shows the representative network for each cluster, i.e. the network which is closest to the centroid of a cluster.



The following table shows the cluster for each graph in SNAP DB

cluster1	wiki-Vote, ca-AstroPh, ca-CondMat, cit-HepPh, cit-Patents, com-dblp.ungraph, flickrEdges, gplus_combined
cluster2	loc-gowalla, amazon0505, amazon0601, as-caida20071105, as-skitter, as20000102, com-youtube.ungraph, email-EuAll, higgs-reply_network, higgs-retweet_network, higgs-social_network, oregon1_010526, oregon2_010526, soc-pokec-relationships, web-BerkStan, web-Google, web-NotreDame, web-Stanford, wiki-Talk
cluster3	p2p-Gnutella31, ca-HepTh, p2p-Gnutella04, p2p-Gnutella25, roadNet-CA, roadNet-PA, roadNet-TX
cluster4	ca-HepPh, ca-GrQc
cluster5	twitter_combined, amazon0302, cit-HepTh, com-amazon.ungraph, com-lj.ungraph, email-Enron, facebook_combined, loc-brightkite, soc-Epinions1, soc-Slashdot0902, soc-sign-Slashdot081106, soc-sign-Slashdot090221, soc-sign-epinions

8. Fitting Graph Models to Real World Networks

We fit each representative network selected in section 7 with model networks of 4 types. The following table enumerates the models, their fitting and generation function.

Kronecker graph	kronfit() from SNAP	krongen() from SNAP
Variation of Nearest Neighbor Model	Analytical model (system of quadratic/linear equations) which solves for parameters u and k. See [9] for details.	Implemented a generator as described in [9]
Forest Fire graph	Publicaly available fitting function [12]	Pubicaly available Generator [12]
Preferential Attachment Model	$\|V\|/\|E\|$	GenPrefAttach() in SNAP Library

After fitting these models, we compute a series of Graph metrics comparing the representative network to its fitted model for each model type. This tells us which type of model best fits a representative network and hence the category of networks it represents. The result with respect to these metrics is show below:

		Euclidian Distance: Target vs Generated Model Graph					Exact Metric Value	
Graph	Models	Sub-Graph Freq	NDD	K_{nn}	dk-2	CC	AS	Diam.
	wiki-Vote						-0.08	3.77
wiki-Vote	Nearest Nbr	6.24e+08	0.23	1687	1187	1.94	0.25	3.87
7,115 nodes	Kronecker	1.51e+09	0.138	2254	1159	2.58	-0.14	3.9
100,762 edges	Forest Fire	7.77e+08	0.21	2651	1306	7.6	-0.1	3.9
	PA	1.1e+09	0.41	1961	4317	2.53	-0.01	2.88
	loc-gowalla						-0.03	5.64
loc-gowalla	Nearest Nbr	7.84e+11	0.069	4243	14873	1.65	0.37	6.4
196,591 nodes	Kronecker	7.16e+11	0.026	3414	28641	2.32	-0.09	3.95
950,327 edges	Forest Fire	5.66e+11	0.039	14853	15603	2.58	0.04	6.89
	PA	7.80e+11	0.43	4320	91183	2.38	-0.01	4.83
	Gnutella31						-0.092	6.72
p2p-Gnutella31	Nearest Nbr	2.19e+08	0.33	500	20626	0.83	0.43	5.76
62,586 nodes	Kronecker	4.27e+09	0.24	1109	21493	0.09	-0.10	3.95
147,892 edges	Forest Fire	3.22e+10	0.27	7996	22325	4.22	0.22	6.98
	PA	1.30e+08	0.60	101	30875	0.035	-0.025	6.52
	ca-HepPh						0.63	5.82
ca-HepPh	Nearest Nbr	4.80e+08	0.096	2748	5208	9.72	0.38	4.5
12,008 nodes	Kronecker	5.12e+08	10.12	2867	5376	10.33	-0.12	3.87
118,521 edges	Forest Fire	4.90e+08	0.097	2756	5178	8.71	0.18	7.47
	PA	5.39e+08	0.37	2948	6261	10.43	-0.04	4.78
	twitter						-0.04	4.54
twitter	Nearest Nbr	3.94e+10	0.053	4534	5984	4.92	0.29	4.42
81,306 nodes	Kronecker	3.89e+10	0.22	3486	11588	6.39	-0.09	3.93
134,2310 edges	Forest Fire	1.05e+12	0.13	44812	13242	14.64	-0.18	4.66
	PA	3.60e+10	0.22	3090	44225	6.43	-0.005	3.57

From the table it is clear that, for all the groups, Nearest Neighbor model is either the best or close to the best model. Kronecker graph is the second best in almost all the cases except for the group represented by "ca-HepPh' graph. For "ca-HepPh" group, forest fire is also close to the best.

The effect of Sub-graph frequency is that it further bolsters the case for Nearest Neighbor model as the best model for all the groups (it was already the case with other metrics).

9. Conclusion

We successfully implemented 2 schemes, [1, 11], for efficiently and accurately counting subgraph frequencies for very large graphs (albeit for 4 node subgraphs only). We used one of these schemes to generate subgraph frequency vectors for most of the graphs in the SNAP Database. We then divided these graphs into group by using K-means clustering and selected a representative network for each group. We then fitted each representative network with 4 types of model graphs. Various graph metrics were computed for these model graphs and compared with those of the representative networks. We found that Nearest Neighbor Model most accurately represent all the groups closely followed by the Kronecker Graph Model.

10. Acknowledgement

I am heavily indebted to Paris Syminelakis for his guidance at every step of this project.

10. References

[1] Subgraph Frequencies: Mapping the Empirical and Extremal Geography of Large Graph Collections [Ugander et. al]
[2] Counting Arbitrary Subgraphs in Data Streams [Kane et. al]
[3] Clique counting in MapReduce: theory and experiments [Finocchi et al]
[4] Fast Approximate Subgraph Counting and Enumeration [Slota & Madduri]
[5] New Hash Functions and Their Use in Authentication and Set Equality [Wegman & Carter] [6] Tabulation Based 4-Universal Hashing with Applications to Second Moment Estimation [Thorup & Zhang]
[7] Universal Hashing and k-wise independent random variables via integer arithmetic without primes [Martin Dietzfelbinger]
[8] Code Repositary - https://github.com/quaizarv/CountingSubgraphs
[9] Measurement-calibrated Graph Models for Social Network Experiments [Sala et al]
[10] Modeling Social Networks through User Background and Behavior [Foudalis et al]
[11] Path Sampling: A Fast and Provable Method for Estimating 4-Vertex Subgraph Counts [Jha et al]
[12] http://current.cs.ucsb.edu/socialmodels/download