Social Network Graph Compression

A BACHELOR'S MINI PROJECT

submitted in fulfillment

of the requirements for the completion of the VI semester

of the

UNDER GRADUATE PROGRAM

in

INFORMATION TECHNOLOGY
(B.Tech in IT)

Submitted by

1. Sourabh Jain(IIT2010038)
2. Aman Kumar(IIT2010043)
3. Prashant Pal(IIT2010042)

Under the Guidance of:

Prof. U.S. Tiwary
IIIT-Allahabad

INDIAN INSTITUTE OF INFORMATION TECHNOLOGY
ALLAHABAD – 211012 (INDIA)

May, 2013

CANDIDATE'S DECLARATION

I hereby declare that the work presented in this project entitled "Social Network Graph Compression", submitted in the partial fulfillment of the completion of the semester VI of Bachelor of Technology (B.Tech) program, in Information Technology at Indian Institute of Information Technology, Allahabad, is an authentic record of my original work carried out under the guidance of Prof. U.S. Tiwary due acknowledgements have been made in the text of the project to all other material used. This semester work was done in full compliance with the requirements and constraints of the prescribed curriculum.

Place: Allahabad
Date:

Sourabh Jain(IIT2010038)

Aman Kumar (IIT2010043)

Prashant Pal (IIT2010042)

CERTIFICATE FROM SUPERVISOR

I do hereby recommend that the mini project report prepared under my supervision by Prof. U.S. Tiwary titled "Social Network Graph Compression" be accepted in the partial fulfillment of the requirements of the completion of VI semester of Bachelor of Technology in Information Technology for Examination

Date:
Place: Allahabad

Prof. U.S. Tiwary
Faculty designation, IIITA

Committee for Evaluation of the Thesis

_____ _____

_____ _____

ACKNOWLEDGEMENTS

We would like to express our greatest gratitude to the people who have helped & supported us throughout our project. We are grateful to our mentor Prof. U.S. Tiwary for his continuous support for the project, from initial advice & contacts in the early stages of conceptual inception & through ongoing advice & encouragement to this day.

A special thank of ours goes to our colleague who helped us in completing the project.

Place: Allahabad
Date:

Sourabh Jain
Aman Kumar
Prashant Pal
B.Tech Final Year, IIITA

ABSTRACT:-

Graphs form the foundation of many real-world datasets.
But the size of graphs presents a big obstacle to understand the essential information they contain. Compressing social networks can substantially facilitate mining and advanced analysis of large social networks. Social network graph must be compressed in away so that they still query able without decompression.
In social network graph query means neighborhood queries.

In this paper we develop an efficient algorithm of compression using eulerian data structure and multi position linearization.

It is an efficient algorithm which can give both in and out neighbor queries efficiently. Compression ratio is also quite good and depends upon the type of graph.

Table of Contents

1. INTRODUCTION

1.1 Currently existing technologies:-

Nowadays, graphs are everywhere ex. social networks, communication networks, biological networks and World Wide Web. A Web graph typically contains a huge number of Web pages as vertices, and an even larger number of hyperlinks as directed edges.

Adler and Mitzenmacher [4] give a web graph compression algorithm by combining vertices which having similar properties or similar connected edges.

Randall et al. [5] use the lexicographic ordering of URLs of Web pages for compressing the graph. Advantage of that is most of the hyperlinks are intra-host and many pages on same page have same hyperlink. So most of the web pages are locally connected to each other , by lexicographical ordering local web pages become consecutive or in neighborhood of each other.

Boldi and Vigna [6,7] further exploited the properties of Web pages in lexicographic ordering to achieve better compression. Specifically, their method takes advantage of the lexicographic locality in Web graphs. That is, proximal pages in URL lexicographic order often have similar neighborhoods. For better compression, Boldi et al. [3] further developed new orderings combining host information and Gray/lexicographic orderings.

Buehrer and Chellapilla [8] used a data mining approach to tackle the problem of compressing Web graphs. Using frequent itemset mining techniques they mined the complete bipartite sub graphs and replaced the edges of those sub graphs by a virtual node connecting to all vertices in both partitions in the complete sub graph, with the combination of gap coding technique in the lexicographic order, achieves the performance of under two bits per edge.

All the method describe above use bits/edge to get primary evaluation measure or considerable factor of performance.

1.2Analysis of previous research in this area:-

We have analysed a greedy technique previously for graph compression given by Fang Zhou[9] . This technique was based on clustering of nodes having similar properties or similar edges.

For a graph , we create a graph summary which consists of a set of vertices (which will contain the nodes after clustering), and a edge correction set which contains two types of edges, positive and negative. Positive edges are those edges which are not present in graph summary but present in original graph. Similarly negative edges are those edges which are present in graph summary but not present in the original graph.

We use a greedy algorithm for implementation. The main idea of the Greedy algorithm is to iteratively group two nodes with the highest cost reduction. The cost reduction is the ratio of the reduction in cost as two vertices are merged into a single node(cluster node). Any two nodes sharing common neighbors can give a cost reduction and the cost reduction achieves its maximum value when all the neighbors of both the nodes are exactly same.

This algorithm has many drawbacks, Practically speaking, it will not answer the neighbor queries in time as the time complexity of the algorithm exceeds O(n^2). Where n is the number of vertices in the original graph.

But when we apply randomization instead of greedy algorithm, it decreases the time complexity of the algorithm but performance gets degraded because greedy algorithm always makes the best compression and consistently produces a smaller compressed graph than the Randomized method does. When the size of the original graph becomes larger, the compression ratio also increases.

This technique supports only in-neighbor queries . For answering out-neighbor queries we have to store the transpose of the graph as well along with the original graph.

1.3 Problem definition and scope:-

We have to given a directed, unweighted friendship relationship graphs of social networks and we want **query-able** compression of these social networks. Social networks should be compressed in a way that they still can be queried efficiently without decompression.

In this paper, we model a social network as a directed graph G = (V,E) where V is a set of vertices and E $\subseteq$ V × V is a set of edges. We also refer to V by V(G) and E by E(G). For an edge e = (u, v), we refer to u as the source of e and v as the destination of e. (u, v) $\neq$ (v, u).

A simple directed graph is a directed graph such that there does not exist a self-loop, i.e., no edge (u, u) for any vertex u, and there is at most one edge from a source u to a destination v. In this paper, we consider simple directed graphs only. It is straightforward to generalize our results and algorithms to deal with directed graphs that contain self-loops and multiple edges between two vertices.

In an undirected graph edges do not carry direction. In a directed graph, edges carry direction.

Notations:-

For an undirected graph G, we can obtain the directed version G $\rightleftarrows$ of G by placing two directed edges (u, v) and (v, u) in G $\rightleftarrows$ for each undirected edge {u, v} in G.

Transpose:

For a graph G, the transpose of G, denoted by GT , is a graph such that V (GT) = V (G) and (u, v) $\in$ E(GT) if and only if (v, u) $\in$ E(G).

Reciprocal:

In a graph G, an edge (u, v) $\in$ E is called reciprocal if (v, u) $\in$ E as well. In such a case, u and v are immediately connected in both directions. Let Fre(G) be the fraction of reciprocal edges in E(G), i.e.,

$$Fre(G) = \frac{\text{number of reciprocal edges in } E(G)}{|E(G)|}$$

Nv = the set of neighbours of v
Ev = $\{\{u_1, u_2\} \in E(\bar{G}) | u_1, u_2 \in N_v\}$

Acc(G) is for a directed graph G we use its underlying undirected graph $\bar{G}$ to define Acc(G), the average clustering coefficient as

$$Acc(G) = Acc(\bar{G}) = \frac{1}{|V(\bar{G})|} \sum_{v \in V(\bar{G})} \frac{2|E_v|}{|N_v|(|N_v| - 1)}$$

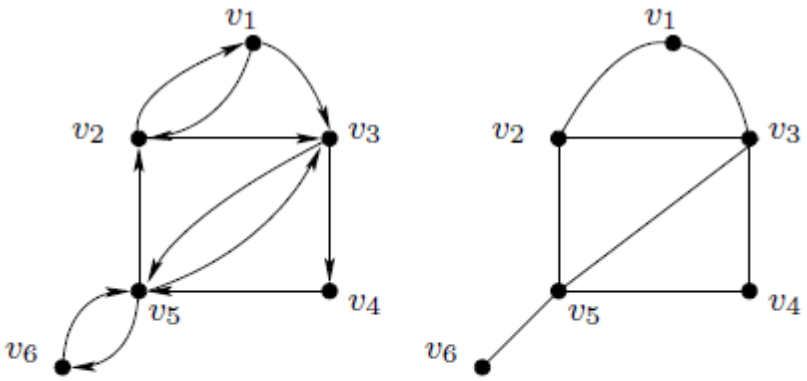
Gcc(G) is the global clustering coefficient :

$$Gcc(G) = Gcc(\bar{G}) = \frac{2 \sum_{v \in V(\bar{G})} |E_v|}{\sum_{v \in V(\bar{G})} |N_v|(|N_v| - 1)}$$

Neighbour Queries in Directed Graphs:

Out-neighbour for a vertex u ∈ V (G), v1 ∈ V (G) is an out-neighbour of u if (u, v1) ∈ E(G).
In-neighbour for a vertex u ∈ V (G), v2 ∈ V (G) is an in-neighbour of u if (v2,u) ∈ E(G).

Neighbour queries –



a) Directed graph G b) The underlying undirected graph $\bar{G}$ of G

$S_1 = (v_6, v_5, v_4, v_3, v_5, v_2, v_3, v_1, v_2)$
$S_2 = (v_6, v_5, v_4, v_3, v_1, v_2, v_5)$
$S_3 = (v_4, v_5, v_3, v_4, v_6, v_5, v_2, v_3, v_1, v_2)$

c) A set of sequences

out-neighbour query on v5 in G returns {v2, v3, v6}.
An in-neighbour query on v5 returns {v3, v4, v6}.
Please note that v3 and v6 are both out-neighbours and in-neighbours of v5, since there are reciprocal edges between v3 and v5 as well as between v5 and v6.

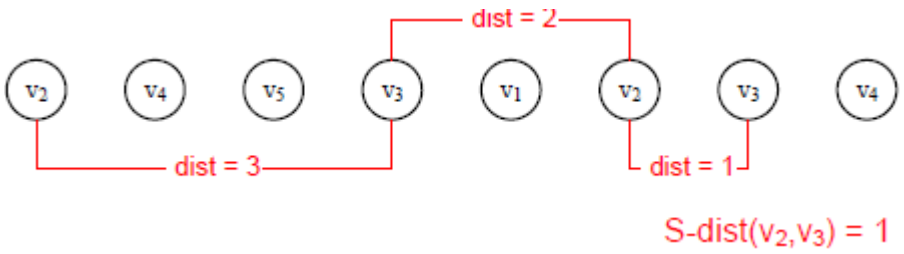
1.4 Formulation of the present problem:

Let S = (vi1 , vi2 , . . . , vim) be a sequence of vertices of graph G (with possible replication). We say S covers G if all the vertices of G appear at least once in S. The length of S is m. Here, S does not need to be a path.

We need the following notion of S-distance.

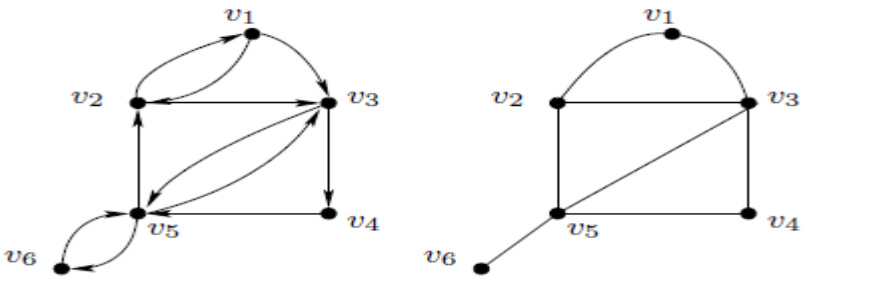
S-distance:-

For a sequence S , the S-distance between u and v, denoted by S-dist(u, v), is the minimum norm-1 distance among all pairs of appearances of u and v.



Multi Position Linearization(MPk):

An MPk linearization of a graph G is a sequence S of vertices of the graph with possible replication, such that S covers G and for all (u, v) $\in$ E(G), S-dist(u, v) $\leq$ k. The length of an MPk linearization is equal to length of S.



a) Directed graph G b) The underlying undirected graph $\bar{G}$ of G

$S_1 = (v_6, v_5, v_4, v_3, v_5, v_2, v_3, v_1, v_2)$
$S_2 = (v_6, v_5, v_4, v_3, v_1, v_2, v_5)$
$S_3 = (v_4, v_5, v_3, v_4, v_6, v_5, v_2, v_3, v_1, v_2)$

c) A set of sequences

In the above figure, the sequences S1 and S3 are two different MP1 linearization of both G and $\overline{G}$. But the length of S1 is less than that of S3. S2 is an MP2 linearization but not an MP1 linearization of G, because for

edge (v5, v3) $\in$ E,S2-dist(v5, v3) = 2.

EULERIAN DATA STRUCTURE:

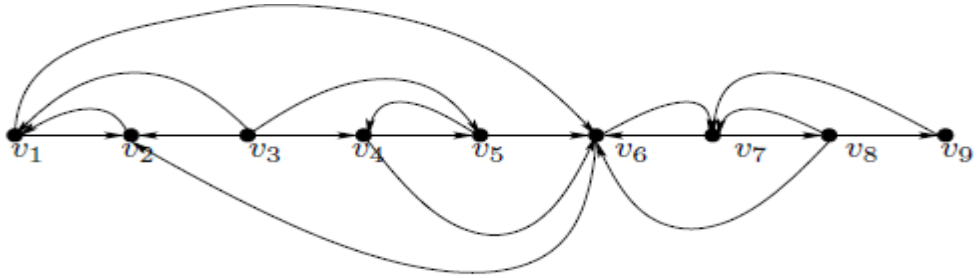
In order to answer both out-neighbour queries and in-neighbour queries efficiently, most of the existing methods have to store both a graph G and its transpose GT .

The Eulerian data structure for a graph G stores an optimal MP1 linearization L of G using an array of the same length as L. Let v(i) be the vertex in G that appears at the position i of L. For cell i of the Eulerian data structure, we keep the following two pieces of information :-

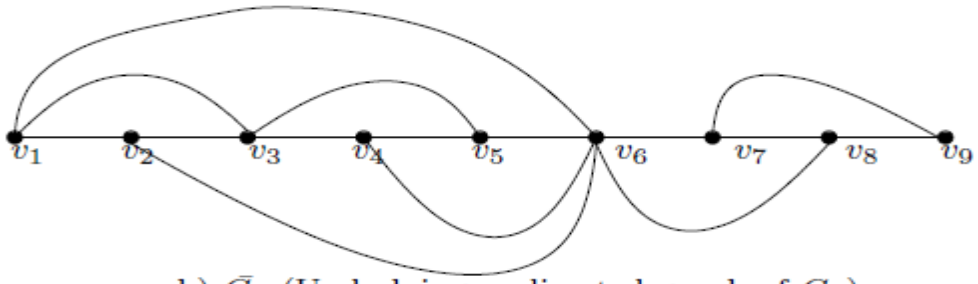
1. **Local information**:- Two bits specifying if edges (V(i-1), V(i)) and (V(i), V(i-1)) belong to E(G), respectively.

2. **Pointer**:- A pointer to the next appearance of v(i). If this is the last appearance of v(i), then the pointer points to the first appearance of the vertex.

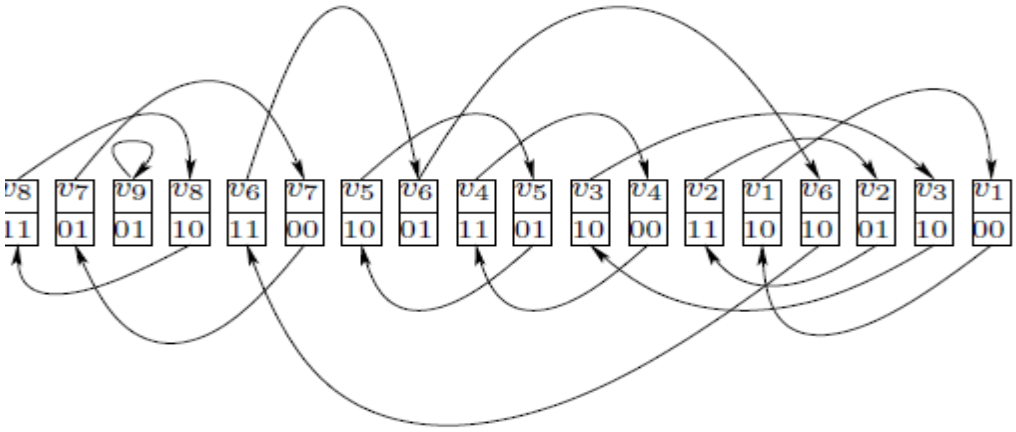
MP linearizations of an arbitrary directed graph:



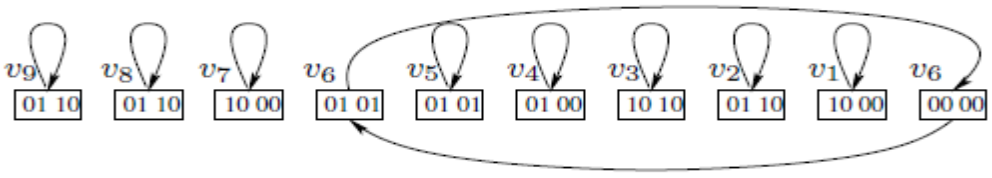
a) Graph G_2



b) $\bar{G}_2$ (Underlying undirected graph of G_2)



c) MP_1 linearization of G_2



d) MP_2 linearization of G_2

The Eulerian data structure of the above shown graph G using an

MP1 linearization is illustrated. Here we show the pointers by arcs. Since the length of the linearization is 18, we need log2(18) = 5 bits to encode each pointer. Therefore, for each position we need 5+2(local information) bits. In total we need 18×(5+2)=126 bits, which make a compression rate of 128/19 ≈ 6.63 bits per edge.

MP2 linearization length of the linearization is 10, we need log2(10) = 4 bits to encode each pointer. Therefore, for each position we need 4+4(local information) bits. In total we need 10×(4+4)=80 bits, which make a compression rate of 80/19 ≈ 4.21 bits per edge.

2. Description of Hardware and Software Used:

2.1 Hardware:

We used a heterogeneous windows 7 based cluster to run most of the experiments. To report the running time, we selected a subsets of our experiments and ran them on a Intel core i3 CPU, 2.40GHz processor, windows 7 base system with 3GB of main memory and 512KB L2 cache memory.

2.2 Software:

The software mainly used for implementing the above graph compression technique using is Dev C++ version 4.9.9.2 which is a standard C++ compiler, on the top of Standard Template Library (STL).

The input file is in text format open able in notepad which consists of multiple lines, each line corresponding an edge in a graph representing source node number and destination node number respectively.

The output file is also in text format open able in notepad or any other text editor which consists of multiple lines, each line consisting of euler data structure (pointer value, vertex number, 2*k bits respectively).

3. Theoretical Tools – Analysis and Development:

3.1 Proposition 1 : (Lower bound of MP1 linearization)

Given a directed graph G, the lower bound for the length of the MP1 linearization of G is :

$$|E(\bar{G})| + 1 = (1 - \frac{Fre(G)}{2})|E(G)| + 1$$

.

Moreover the bound is tight if and only if $\overline{G}$ has an Eulerian path, i.e. it has at most two vertices of odd degrees. Eulerian path construction problem which is an existence problem, finding the optimal MP1 linearization is an optimization problem.

Lemma 1 : The minimum length of MP1 linearization of an arbitrary directed graph G is $|E(\overline{G})| + \max\{n_{odd}/2, 1\}$, where n_{odd} is the number of vertices with odd degrees in $\bar{G}$.

Proof :

In any undirected graph $\overline{G}$, the sum of degrees of all vertices is even. Thus, the number of vertices with odd degrees is also even. We first prove by induction that for any graph G we can obtain an MP1 linearization of length $|E(\overline{G})| + \max\{nodd/2, 1\}$.

Basis: For $n_{odd} = 0$ and $n_{odd} = 2$, the claim follows from proposition .

Induction:

Since n_{odd} must be even, we assume that the lemma holds for $n_{odd} = 2k$ ($k \geq 1$), and consider the case when $n_{odd} = 2(k + 1)$. There are two sub-cases.

First sub-case, there are two vertices u and v with $\bar{G}$ odd degrees such that they are not connected in. We add an edge {u, v} to $\bar{G}$ and call the new graph $\bar{G}_*$.

Since $\bar{G}_*$ has only 2k vertices with odd degrees, applying the induction assumption, we can obtain an MP1 linearization of $\bar{G}_*$ with length $|E(\bar{G}_*)| + k = |E(\bar{G})| + k + 1$. Since $E(\bar{G}) \subset E(\bar{G}_*)$ the MP1 linearization of $\bar{G}$ is indeed an MP1 linearization for $\bar{G}$.

In the second sub case, all vertices with odd degrees are connected. We arbitrarily take two vertices u and v with odd degrees and remove the edge {u, v} from $\bar{G}$. Let us call the resulting graph $\bar{G}_*$. $\bar{G}_*$ has 2k vertices with odd degrees. Therefore, there is an MP1 linearization with length $|E(\bar{G}_*)| + k = |E(\bar{G})| - 1 + k$ for $\bar{G}_*$. Since the MP1 linearization for $\bar{G}_*$does not cover {u, v}, we have to add u and v to the end of the linearization. Therefore we just build an MP1 linearization for $\overline{G}$ of length |E$\bar{G}$)| + k + 1, as desired.

3.2Theorem: An Eulerian data structure to encode a graph G uses up to

$$(1 - \frac{Fre(G)}{2} + \frac{1}{\bar{d}})\Big(\lceil \log_2(|V(\bar{G})|) + \log_2(\bar{d}+1)\rceil + 1\Big)$$

bits per edge on average, where $\bar{d}$ is the average degree of $\bar{G}$.
Moreover, using this data structure, it is possible to answer the in-neighbor and out-neighbour queries for any vertex v in

$$O(\textstyle\sum_{u \in N_v} deg(u) \log |V(G)|)$$

time, where deg(u) is the degree of vertex u in $\bar{G}$ and Nv is the set of out-neighbours/in-neighbours (resp.) of v in out-neighbour/in-neighbour queries.

Proof:

Let L be an optimal MP1 linearization of $\bar{G}$ (therefore for G as well). Since there are at most |V | vertices of odd degrees in $\bar{G}$, the upper bound for the length of L is

|E($\bar{G}$)| + |V ($\bar{G}$)|/2.

Using 2 bits to store the local information and log$_2$(|E($\bar{G}$)| + |V ($\bar{G}$)|/2) bits for the pointer for each cell, in total the Eulerian data structure uses at most

$$\Big(|E(\bar{G})| + |V(\bar{G})|/2\Big)\Big(2 + \lceil \log_2(|E(\bar{G})| + |V(\bar{G})|/2)\rceil\Big)$$

bits. To get the bits/edge rate we divide this by the number of edges of G, and have

$$\Big(\frac{|E(\bar{G})| + |V(\bar{G})|/2}{|E(G)|}\Big)\Big(2 + \lceil \log_2(|E(\bar{G})| + |V(\bar{G})|/2)\rceil\Big)$$

Notice that we can write the ratio of |E($\bar{G}$)|/|E(G)| in terms of Fre(G) and also

$$\frac{|V(\bar{G})|}{2|E(G)|} \leq \frac{|V(\bar{G})|}{2|E(\bar{G})|}$$

which is precisely the inverse of the average degree of $\bar{G}$.

Therefore, the bits/edge rate is at most:

$$\left(1-\frac{Fre(G)}{2}+\frac{1}{\bar{d}}\right)\left(2+\lceil\log_2(|E(\bar{G})|+|V(\bar{G})|/2)\rceil\right)$$

We use $|E(\bar{G})|=\frac{|V(\bar{G})|\bar{d}}{2}$ to further simply the inside of the logarithm and obtain

$$\left(1-\frac{Fre(G)}{2}+\frac{1}{\bar{d}}\right)\left(2+\lceil\log_2(|V(\bar{G})|\bar{d}/2+|V(\bar{G})|/2)\rceil\right)$$
$$=\left(1-\frac{Fre(G)}{2}+\frac{1}{\bar{d}}\right)\left(2+\lceil\log_2(|V(\bar{G})|)+\log_2(\bar{d}+1)-\log_2(2)\rceil\right)$$

,

The upper bound is proved.
Since each vertex u of G appears in at least one position in the Eulerian data structure, we use the first position of u in L as the identifier for the vertex. Therefore, for an out-neighbour/in-neighbour query on vertex u, we have to return the positions of the first appearances of all out-neighbours/in-neighbours of u.

- local information (only two bits) for each position takes constant time.
- The pointer takes O(log |V (G)|) time (the number of bits for each pointer).

Since the length of the linked list of positions for a vertex u is log(deg(u)) in the MP1 linearization of G, traversing over the linked list takes $O(\deg(u)\log_2|V(G)|)$ time.
By traversing the linked list of u we can retrieve the positions of all neighbours of u. However, for a neighbour v of u, the retrieved position may not be the first appearance of v. Therefore, for each retrieved neighbour v, we have to traverse the linked list for v to get the first appearance.

So, answering an out-neighbour/in-neighbour query takes $O(\log(|V(G)|)\sum_{u\in N_v}deg(u))$ time in total.

As a baseline for representing a graph G with sub-linear in-neighbour and out-neighbour queries, we can use 2log2 |V (G)| bits to encode an edge.

For social networks in practice, it is reasonable to assume that the average degree increases logarithmatically with respect to the number of nodes in the graph.

Therefore, asymptotically (i.e., assuming the number of nodes approaches infinity) the Eulerian data structure uses half of the number of bits that the baseline schema uses due to the following equation.

$$\lim_{|V(G)|\to\infty} \frac{(1+\frac{1}{\log_2|V(G)|})(\log_2(|V(G)|)+\log_2\log_2(|V(G)|)+1)}{2\log_2(|V(G)|)} = \frac{1}{2}$$

To the best of our knowledge, the Eulerian data structure is the first schema that allows answering both out-neighbour and in-neighbour queries in sub linear time, and provides a nontrivial theoretical upper bound on the number of bits per edge. The upper bound given by the above Theorem is for arbitrary graphs, including totally random graphs.

We know that for some subclasses of graphs the information theoretic lower bound is log(N) bits per edge.

Therefore, from a theoretical point of view our upper bound is not very far away from this information theoretic lower bound, at least asymptotically close.

4. Development of Algorithm:

4.1 MPk Algorithm:-

First we give an algorithm to determine optimal MP1linearization algorithm and then we extend the idea up to MPk linearization.

Finding Optimal MP1 linearization algorithm:

- Start from a node with odd degree, if there is no such a node, start from an arbitrary node with nonzero degree.
- Choose an edge whose deletion does not disconnect the graph, unless there is no other choice.
- Move across the edge and remove it.
- Keep removing edges until getting to a node that does not have any remaining edge to choose.
- If the graph is not empty go to step 1.

This algorithm partitions the edges to exactly Nodd/2 edge-disjoint paths, where Nodd is the number of vertices with odd degree (assuming Nodd > 0). The length of an optimal MP1 linearization is |E| + Nodd/2 (Nodd > 0).

It can be implemented in O(|E|) time.

A natural extension for the Eulerian data structure is to use MP_k linearization instead of MP1 linearization. This raises several new challenges. First of all, unlike MP1 linearization, finding an optimal MPk linearization for $k \geq 2$ is NP-hard in general, since it can be regarded as a generalization of the minimum bandwidth problem.

Given an MP_k linearization, we need to store 2k bits as the local information for each position i to record whether (i, j) and (j, i) are edges in the graph for $|i-j| \leq k$. The amount of local information is considerable. Hence, storing the local information efficiently is important.

Two sub-problems:

1. MP_k linearizing a graph
2. Encoding the local information for each position of the MP_k linearization.

To ensure that we can handle large social networks, we use a straightforward greedy for linearizing a graph.

MPk linearization greedy algorithm[10]:

- We start with a random vertex.
- At each step we append to the list the vertex that has the largest number of edges with the last k nodes in the list.
- We remove these edges from the graph and iterate until no edge is left.
- If none of the last k vertices in the list have a neighbour, and graph is not empty go to step 1.

Using 2k bits we can encode the local information for each position. A practical problem of the greedy linearization heuristic is that, as we are removing the edges of the graph, the graph becomes sparser and sparser.

Thus, having a fixed k all the time is not a good idea since the rear part of the linearization may have very few new edges to encode.

To be adaptive, we use a relaxed version of the linearization notion. We start with a relatively large value of k and watch.

Algorithm 1 Find an MP_K linearization of G

Input: K, reducing factor RF $(0 \leq RF \leq 1)$, density threshold DT $(0 \leq DT \leq 1)$ and Graph G
Output: Linearization L of G
Description:

```
1: initialize L to an empty list
2: while |E(G)| ≥ 1 do
3:    let u be a random node with nonzero degree
4:    append u to L
5:    /* let X be the set of the last K vertices in L */
6:    while X has at least one neighbor in V(G) − X do
7:       let v be the node which has the most number of
         edges to and from X
8:       remove all edges between v and vertices in X
9:       edgecount+ = deg_old(v) − deg_new(v)
10:      append v to L
11:      if Length(L)%1000==0 then
12:         if edgecount/2 * K * 1000 < DT then
13:            K = K * RF
14:         end if
15:         edgecount = 0
16:      end if
17:   end while
18: end while
```

the average local density for the recent positions in the list (the last 1000 positions as shown in Algorithm 1). Once it drops below a certain density threshold DT, we reduce k by multiplying it to a predefined reducing factor RF.

We choose to use a simple heuristic for linearization and encode the local information.

We can see as we remove edges graph become sparse, and now we can decrease the value of k so that total local information in euler data-structure decreases.

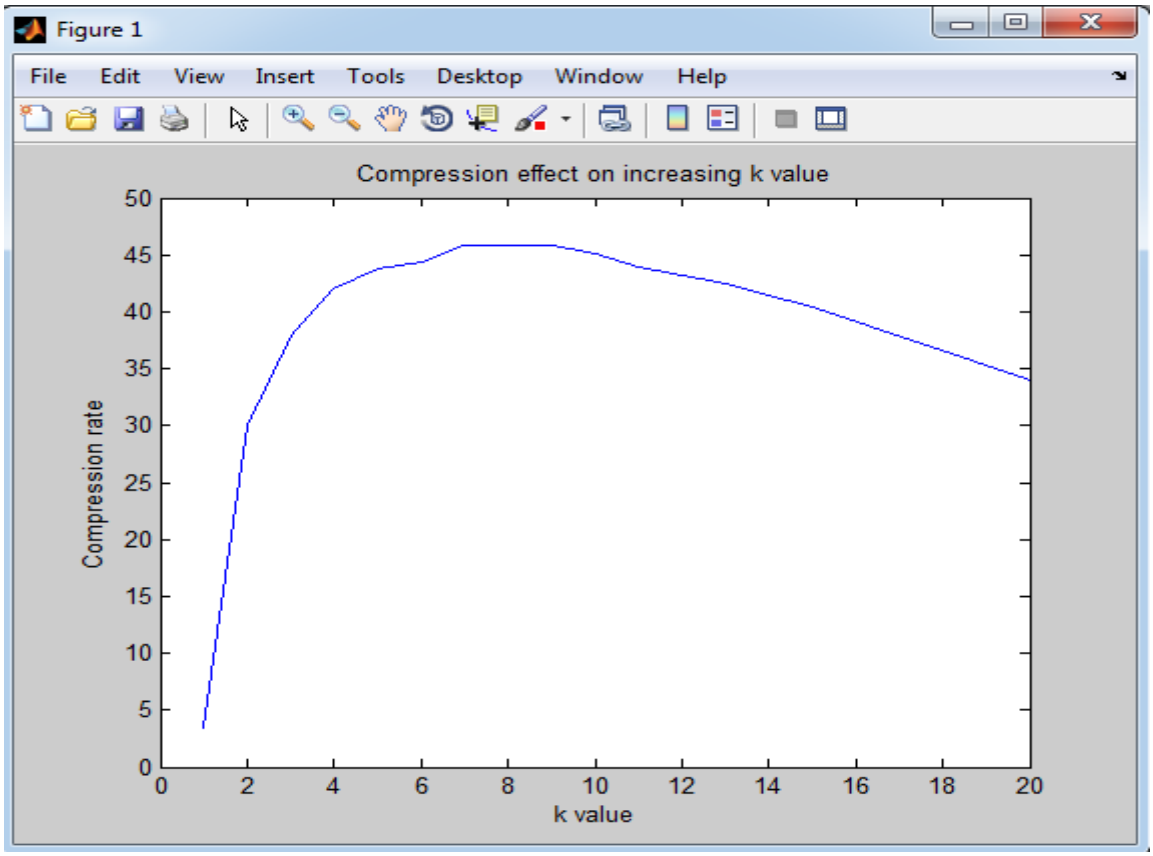
Decrease in value of k is decided using Reducing factor (RF) and Density Threshold (DT). In algorithm when the average local density for the recent positions in the list (the last 1000 positions as shown in Algorithm 1). Once it drops below a certain density threshold DT, we reduce k by multiplying it to a predefined reducing factor RF.

5. Testing and Analysis:

5.1 Testing:

We use multiple dataset for test our algorithm and obtain different result of compression depending upon the characteristic of graph. For deferent value of Fre (G) we obtain deferent compression ratios.
Compression ratio also depends upon k value which we choose for linearization.

We use a peter-hamster dataset for testing our algorithm for different k values. That graph was having around 22,900 edges with 2400 vertices.
and Fre(G) = 0.55
For different k values we obtain different compression ratios.



Here we can see as we increase the value of k compression ratio increases but after a certain value it start decreasing.
Here maximum compression in on k=8 around 46.2% without using heuristic.

Result of Heuristic Testing on different Datasets :

Name	Description	V(G)	E(G)	Fre(G)
peter-hamster	Social ntwork datasets of peter-hamster	2422	22962	0.55

(k , RF)	(10,1)	(10 , 0.9)		(15,0.9)	
DT	0	0.15	0.25	0.15	0.25
peter-hamster	45.1	47.4	47.54	46	46.3

Different type of threshold values and redundancy factor we obtain different compression rate.

5.2Analysis:

There is a trade-off between the length of linearization against the amount of local information. The trade-off highly depends on the structure of the graph.

Intuitively, for a large sparse graph where each vertex has the same out degree, and the destinations of the out edges are picked randomly, increasing k would not influence the length of the linearization significantly.

However, for a large random dense graph G where the existence of an edge from every node to another is independently determined by a probability of 50%, increasing k up to |V (G)|, the number of vertices, is actually beneficial.

In social networks, the number of vertices with a small degree is much more than the number vertices with a large degree. Consequently, average clustering coefficient usually has a bias toward the vertices with small degrees. Therefore, we consider the measure Global clustering coefficient (Gcc). Gcc measures the probability that there is an edge between two vertices when they have a common neighbour. Consider a social network with a large Gcc value. Suppose vertices u, v, w and t are four consecutive vertices in an MPk linearization, and there are edges between u and v, v and w, as well as w and t. There is a good chance that u and w are connected, since both have v as a neighbour. Moreover, if u and w are connected, using the same argument again, there is a good chance that u and t are also connected. Depending on how strong the locality of the network is, it does make sense to keep more bits for u specifying weather {u, w}, {w, u}, {u, t} and {t, u} belong to E($\bar{G}$) or not.

6. CONCLUSIONS:

In this paper, we tackled the problem of compressing social networks in a neighbour query friendly way. We developed an effective social network compression approach achieved by a novel Eulerian data structure using multi-position linearizations of directed graphs.

Importantly, our method comes with a nontrivial theoretical bound on the compression rate. To the best of our knowledge, our approach is the first that can answer both out-neighbour and in-neighbour queries in sub-linear time.

- We introduce a novel framework for representing graphs.
- In its simplest settings, our framework comes with an upper bound on bits/edge rate.
- Our method can efficiently answer more types of query and retain the comparable compression rate than the state-of-the-art methods.
- Our method can respond both in-neighbour queries and out-neighbour queries without storing transpose of graph.
- Our framework reduces the problem of compressing a graph to an intuitive combinatorial problem

7. Future Work:

The encouraging results in this study suggest several interesting future directions.

- It is interesting to explore approximation methods for MPk linearization for k>1.
- It is interesting to explore effective methods to determine a good value of k for MP_k linearization compression of social networks. Last, our heuristic algorithm is simple.
- Finding smarter algorithms for linearizing a graph.
- Using other compression techniques on top of our framework.
- Hardness result for MPk linearization when k is fixed.

It leaves space for further improvement in both the compression rate and the compression runtime.

APPENDIX:

Explanation of the Source Code:

We have two source program in C++, one for compression and other one for decompression.

In compression we have implemented MPk algorithm. In which value of k is input by the user and input of graph is done by "input.txt" and for that it will generate "output.txt" .

```
1.  cin>>k;
2.  while(edge>=1)
3.  {
4.          int u=get_random_vertex();
5.          push_vertex(u);
6.          create_map();
7.          while(edge>=1)
8.          {
9.                  if(m.size()==0)
10.                         break;
11.                 else
12.                 {
13.                         int v=get_max_vertex();
14.                         remove_edge_all(v);
15.                         update_map_decrease();
16.                         push_vertex(v);
17.                         update_map_increase(v);
18.                 }
19.
20.         }
21.         m.clear();
    }
```

It is the main function in compression program.
Line 1: we are taking input from user.

Line 2: is a loop which will break when there are no edge in graph remaining.

Line 4: using a module get_random_vertex() which will give a radom vertex which have at least one edge to or from another vertex .

Line 5: Now using a module push_vertex() which push euler structure of that generated random vertex into sequence which we want in result and set it's pointer to accordingly and having s 2*k null.

Line 6: Use a module create_map() that will store information about last k entries in sequence into map data structure.

Line 7: Loop until graph is empty.

Line 9: If map is Null or last k vertices have no edge in graph loop will break.

Line 13: Otherwise use module get_max_vertex() , which uses the value of map and generate the vertex(no in last k vertices of sequence) which have max edges to or from last k vertices of sequence.

Line 14: using module remove_edges_all(), which will remove all the edges between latest generated vertex and last k vertices of sequence and also update(append) the local information of last k vertices.

Line 15: using module update_map_decrese(), which will update map which was creating initially by removing the effect of currently deleted edges and effect of previous kth vertex which now is not in k vertices.

Line 16: using again module push_vertex().

Line 17: use module update_map_increase(), which will update the value of map for the vertex which is appended in sequence currently at line 16.

Line 21: clear map is use to erase all values from map.

Memory variables:

```
// Memory(Variables)
int info[2500][2];
set<int> s[2500];
set<int> s_rev[2500];
set<int> ver;
map<int,int> m;
vector<node> ans;
int k,edge;
```

Modules:

1. get_random_vertex():

```
int get_random_vertex()
{
        set<int>::iterator it;
        it=ver.begin();
        int p=rand()%ver.size();
        while(p--)
                it++;
        return *it;

}
```

2. push_vertex():

```
void push_vertex(int u)
{
        if(info[u][0]==-1)
        {
                info[u][0]=ans.size();
                info[u][1]=ans.size();
        }
        else
        {
                ans[info[u][0]].pointer=ans.size();
                info[u][0]=ans.size();
        }
        node temp;
        temp.pointer=info[u][1];
        temp.vertex=u;
        temp.s="";
        ans.pb(temp);
}
```

3. create_map():

```
void create_map()
{

    int i=max(0,ans.size()-k);
    while(i<ans.size())
    {
            int p=ans[i].vertex;
            set<int>::iterator it;
            for(it=s[p].begin();it!=s[p].end();it++)
                    m[*it]++;
                      //cout<<*it<<endl;
            for(it=s_rev[p].begin();it!=s_rev[p].end();it++)
                    m[*it]++;

            i++;
     }


}
```

4. remove_edges_all():

```
oid remove_edge_all(int v)
{
    for(int i=max(0,ans.size()-k);i<ans.size();i++)
    {
        int w=ans[i].vertex;
        set<int>::iterator it;
        set<int>::iterator it1;
        it=find(s[w].begin(),s[w].end(),v);
        if(it!=s[w].end())
        {
            s[w].erase(it);
            ans[i].s+="1";
            // delete same from its reverse set
            it1=find(s_rev[v].begin(),s_rev[v].end(),w);
            s_rev[v].erase(it1);
            if(s[w].size()==0)
            {
                it1=find(ver.begin(),ver.end(),w);
                    ver.erase(it1);
            }
            edge--;
        }
        else
            ans[i].s+="0";
        it=find(s_rev[w].begin(),s_rev[w].end(),v);
        if(it!=s_rev[w].end())
        {
            s_rev[w].erase(it);
            ans[i].s+="1";
            it1=find(s[v].begin(),s[v].end(),w);
            s[v].erase(it1);
            if(s[v].size()==0)
            {
                it1=find(ver.begin(),ver.end(),v);
                ver.erase(it1);
            }
            edge--;
        }
        else
            ans[i].s+="0";
    }
}
```

5. update_map_decrease():

```
void update_map_decrease()
{
        set<int>::iterator it1;
        map<int,int>::iterator it2;
        int i=ans.size()-k;
        if(i<0)
                return ;
        int u=ans[i].vertex;
        for(it1=s[u].begin();it1!=s[u].end();it1++)
                m[*it1]--;
        for(it1=s_rev[u].begin();it1!=s_rev[u].end();it1++)
                m[*it1]--;
       // print_map();
        it2=m.begin();
        while(it2!=m.end())
        {
                if(it2->second==0)
                {
                        m.erase(it2);
                        it2--;
                }
                else
                        it2++;
        }
}
```

6. update_map_increase():

```
void update_map_increase(int v)
{
      set<int>::iterator it;
      for(it=s[v].begin();it!=s[v].end();it++)
                  m[*it]++;
      for(it=s_rev[v].begin();it!=s_rev[v].end();it++)
                  m[*it]++;

}
```

Second Source program for decompression which uses the “output.txt” generated by compression program and generate “decompress_output.txt” which having exactly number of edges as “input.txt”.
This program traverse each vertex of original graph only once and uses k value to check whether last and next k vertices in sequence connected to it, using local information stored in euler data structure, and create an edge and insert into decompressed graph and move using pointer value.

Working Source code of decompression:

```
for(int i=0;i<s;i++)
    {
        if(!arr[i])
        {
            int ct=0,j=i,t;
            while(ct==0)
            {
                t=2;
                for(int p=1;p<=k;p++)
                {
                    if(j-p>=0 && seq[j-p].st.length()>=t)
                    {
                        if(seq[j-p].st[t-1]=='1')
                            fout<<seq[j].vertex<<" "<<seq[j-p].vertex<<endl;
                    }
                    else
                        break;
                    t+=2;
                }
                t=1;
                for(int p=1;p<=k;p++)
                {
                    if(j+p<s && seq[j].st.length()>=t)
                    {
                        if(seq[j].st[t-1]=='1')
                            fout<<seq[j].vertex<<" "<<seq[j+p].vertex<<endl;
                    }
                    else
                        break;
                    t+=2;
                }
                arr[j]=true;
                if(seq[j].pointer<=j)
                  ct++;
                j=seq[j].pointer;
            }
        }
    }
```

Heuristic code:

```
1.  if(ans.size()%1000==0)
2.  {
3.          if ((edgecount*1.0)/(2.0*k*1000.0) <dt)
4.          {
5.                  k=k*rf;
6.                  break;
7.          }
8.          else
9.              edgecount=0;
10. }
```

Here Redundancy factor is rf and density factor is df, which is input by user and value of k is decreased by multiple of rf.

References:

[1]. F. Chierichetti et al. , ACM-KDD Cup, 2009, "On compressing social networks",

[2]. "Jérôme Kunegis. KONECT - the Koblenz Network Collection". Online konect.uni-koblenz.de, 2013.

[3]. P. Boldi, M. Santini, and S. Vigna. Permuting web graphs, , Berlin, Heidelberg, 2009 , In Proceedings of the 6th International Workshop on Algorithms and Models for the Web-Graph (WAW'09).

[4]. M. Adler and M. Mitzenmacher, In Data compression conference: 2001, "Towards compressing web graphs".

[5]. K. H. Randall et al., In DCC :2002, "The link database: Fast access to graphs of the web".

[6] P. Boldi and S. Vigna, WWW :2004, "The web-graph framework I: compression techniques".

[7] P. Boldi and S. Vigna, In Data Compression Conference :2004, "The web-graph framework II: Codes for the world-wide web".

[8]. G. Buehrer and K. Chellapilla. In WSDM :2008, "A scalable pattern mining approach to web graph compression with communities".

[9]. Fang Zhou, In Data compression conference :2009, "Graph Compression".

[10]. Hossein Maserrat and Jian Pei, In ACM-KDD Cup :2010, "Neighbor Query Friendly Compression of Social Networks".