Bitcoin: A Complex Network perspective

Alok Sharma
15MA40002

Supervisor: **Prof. Bibhas Adhikari**

M.Sc. project report - 4th Semester.

Department of Mathematics
Indian Institute of Technology, Kharagpur
India

April, 2017

Certificate

This is to certify that the project titled **Bitcoin: A Complex Network Perspective**, submitted by Alok Sharma, Roll No: 15MA40002 in partial fulfillment of the degree of M.Sc. in Mathematics to the Indian Institute of Technology Kharagpur, is a bonafide record of the work carried out under my supervision and guidance. This report fulfills all the requirements as per regulations of the institute and has not been submitted to any other institute/university for any degree or diploma.

Prof. Bibhas Adhikari
Department of Mathematics
Indian Institute of Technology, Kharagpur.

Acknowledgement

First of all, I would like to thank Prof. Bibhas Adhikari for consenting to supervise me for my M.Sc Thesis and providing me with the optimal environment to work under him. I express my sincere gratitude to him for taking into account my research interests and showing me the way forward. I would also like to thank my parents and friends for their enormous support.

Contents

Abstract

The Bitcoin system is unique in many network theoretic aspects. It is highly temporal, it grows really quickly, but at the same time, it responds nicely to classic algorithms like PageRank. In the course of this study, we computed various network-theoretic statistics on multiple snapshots of the Bitcoin network and tried to analyse it using these statistics. We also develop and document general purpose python modules to compute these statistics over any desired snapshot of the Bitcoin network. We also present the BitRank algorithm which takes inspiration from classic PageRank but 'appears' to be more effective in finding interesting nodes in the Bitcoin network.

Chapter 1

Introduction

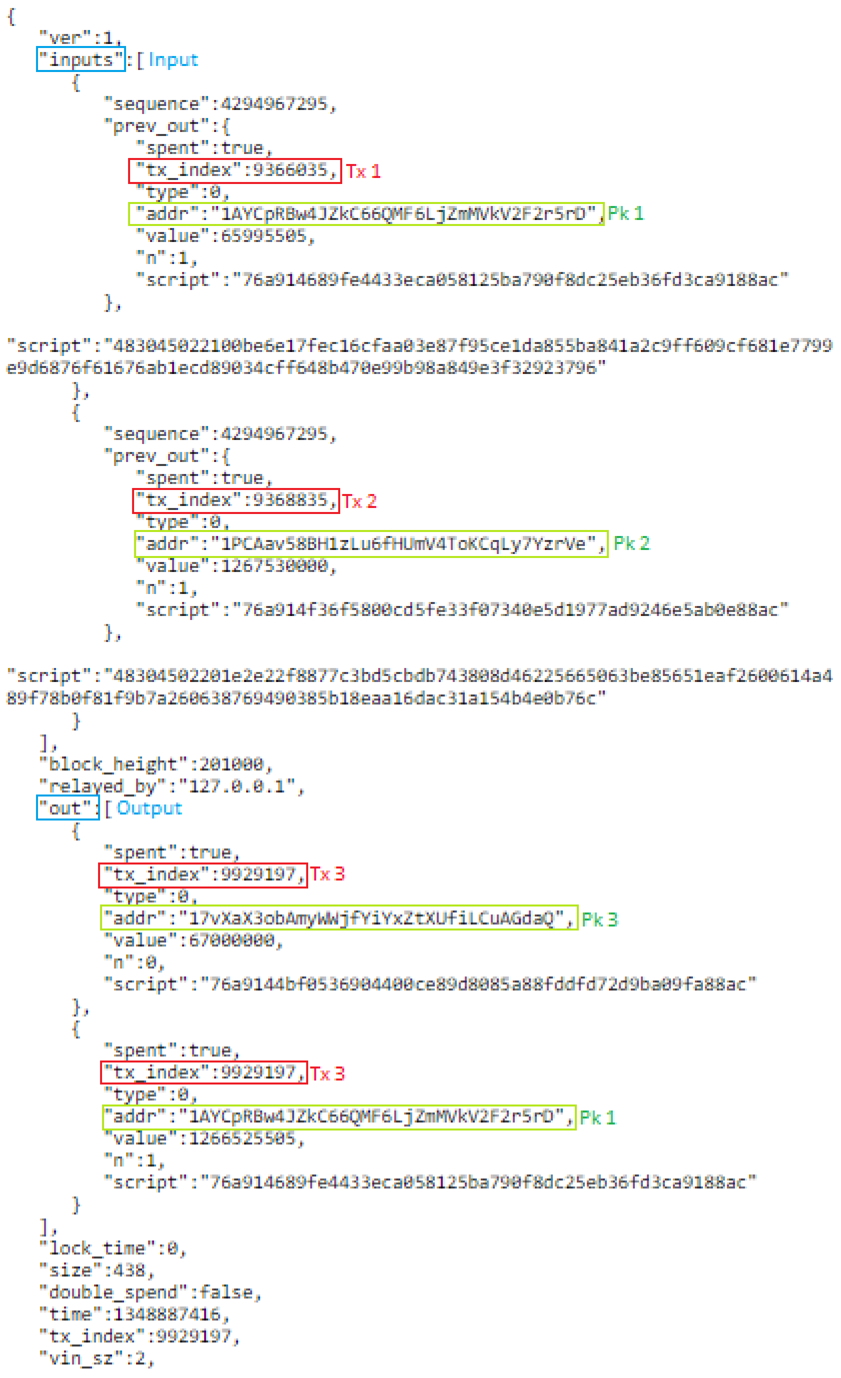
Bitcoin [Nak08] is a decentralized anonymity preserving electronic cryptocurrency. The scheme was developed by Satoshi Nakamoto(a pseudonym) and started in January 2009. It has attracted lot of media attention since then. This use of blockchain technology to create a tamper resistant database is opening many new fronts in the way data is stored, viewed and manipulated. The current value of 1 BTC is approximately 1200US$. The website [10] provides the complete documentation of the Bitcoin technology and developer reference. Book [Fra14] provides a more comprehensive discussion of the security protocols of Bitcoin and associated cryptocurrencies.
Throughout our study, we have used Tx and PK to denote a Transaction and Public Key pver the Bitcoin network respectively.
Bitcoin uses the elliptic curve algorithm[JMV01] with the DSA signature scheme. Interest in Bitcoin system has been primarily due to its cryptographic and anonymity preserving protocols. One notable study [RH13] followed by [HF16] shows how network analysis may catch patterns in the network and compromise the anonymity provided by the system. [Nic15] provides an extensive discussion of possible algorithms to break the anonymity. But all these studies take into account the Multi-Input Heuristic. In fact, [HF16] attempts to completely explain the effectiveness of this algorithm. After application of the Multi-Input Heuristic, the Public Keys (PKs) which occur in the input line of the same transation are affiliated to a single ”user”. This is reasonable since to create a transaction with two PKs, the user needs to know the corresponding ”secret” key of both the Public Keys. Since, the secret key is a confidential information and is secured via high cryptographic standards, we have reason to believe that a user in possession of both the secret keys must also imply that he/she owns the corresponding Public Keys as well.
[FKP15] also attempted to quantitatively analyse the Bitcoin Network but most of their analyses was performed on a day’s data. [RS13] downloaded

the full history of Bitcoin Network till block at height 180,000 and computed various statistics over it. We, on the other hand, take a snapshot of the network and then analyze that snapshot. Our dataset has 5000 blocks but the generated graph has 1.2 million unique Public Keys (PKs) which is comparable to [RH13], [FKP15] and is 1/3rd the size of [RS13]. This enormous increase in volume of newly generated Public Keys from 2011 has been duly noted by [HF16]. Note however that PKs are just aliases using which, the actual users hide their identity. To capture this two-faced behavious, we consider our networks in two ways, on one way, as a simple graph and on the other way, as a directed multigraph. The simple graph like structure captures social-tie information while the multigraph captures the complex nature of the network. We discuss the distinction in the coming chapters.
[HF16] and [FKP15] both note that even though the network is highly temporal, applying traditional algorithms (like PageRank[Pag+99][BP12]) still produces consistent results and highlights the most interesting nodes. These observations have further peaked the interest in the study of this network.
We also did an exercise to compute the various Motifs this network has. Note that the Tx Network is a Directed Acyclic Graph but the PK network can have any kind of structure. We used the Transaction types to classify the possible Motifs in this network.
We have used Python to create modules which could be run independently to gather various statistics over the Bitcoin Network.
The network was stored as .dat files inside Bitcoin Core (The official Bitcoin Wallet). [Gar16] was forked and used to interact with Bitcoin core to get the desired data in appropriate format. Due to the huge volume of data (approx 80GB), information was "trimmed" and only the bare essential was kept.
We also developed our version of the PageRank algorithm, called BitRank, and to implement it we created another version of the Bitcoin network, this time using weighted data structures. The edge weights were decided in accordance with the amount of bitcoins being transferred between Public Keys. More description can be found in the coming section.
[LS16] was used with Python to perform many network related tasks.
We now define the types of networks we have considered over Bitcoin.

1.1 Types of Networks

Figure 1.1 shows a sample 2-Input-2-Output Transaction (Tx) over the Bitcoin Network (courtesy: https://Blockchain.info [16]) The Input section

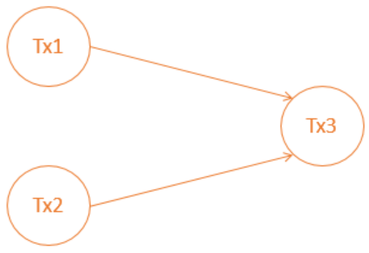
contains the Transaction Ids (TxIds) from which the current Transaction is claiming money. In this example, the input Tx are Tx1 and Tx2. The input addresses which are being claimed are Pk1 and Pk2.
The Output section contains the TxId of the current transaction in both the output TxId fields. It is shown as Tx3. This section also contains 2 Public Keys to which the money is being sent to. One of these is incidentally same as one of the PKs in the Input field.



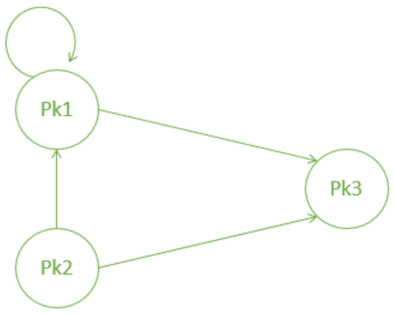
```
{
   "ver":1,
   "inputs":[ Input
      {
         "sequence":4294967295,
         "prev_out":{
            "spent":true,
            "tx_index":9366035, Tx 1
            "type":0,
            "addr":"1AYCpRBw4JZkC66QMF6LjZmMVkV2F2r5rD", Pk 1
            "value":65995505,
            "n":1,
            "script":"76a914689fe4433eca058125ba790f8dc25eb36fd3ca9188ac"
         },

"script":"483045022100be6e17fec16cfaa03e87f95ce1da855ba841a2c9ff609cf681e7799
e9d6876f61676ab1ecd89034cff648b470e99b98a849e3f32923796"
      },
      {
         "sequence":4294967295,
         "prev_out":{
            "spent":true,
            "tx_index":9368835, Tx 2
            "type":0,
            "addr":"1PCAav58BH1zLu6fHUmV4ToKCqLy7YzrVe", Pk 2
            "value":1267530000,
            "n":1,
            "script":"76a914f36f5800cd5fe33f07340e5d1977ad9246e5ab0e88ac"
         },

"script":"48304502201e2e22f8877c3bd5cbdb743808d46225665063be85651eaf2600614a4
89f78b0f81f9b7a260638769490385b18eaa16dac31a154b4e0b76c"
      }
   ],
   "block_height":201000,
   "relayed_by":"127.0.0.1",
   "out":[ Output
      {
         "spent":true,
         "tx_index":9929197, Tx 3
         "type":0,
         "addr":"17vXaX3obAmyWWjfYiYxZtXUfiLCuAGdaQ", Pk 3
         "value":67000000,
         "n":0,
         "script":"76a9144bf0536904400ce89d8085a88fddfd72d9ba09fa88ac"
      },
      {
         "spent":true,
         "tx_index":9929197, Tx 3
         "type":0,
         "addr":"1AYCpRBw4JZkC66QMF6LjZmMVkV2F2r5rD", Pk 1
         "value":1266525505,
         "n":1,
         "script":"76a914689fe4433eca058125ba790f8dc25eb36fd3ca9188ac"
      }
   ],
   "lock_time":0,
   "size":438,
   "double_spend":false,
   "time":1348887416,
   "tx_index":9929197,
   "vin_sz":2,
```

Figure 1.1: A sample Transaction (Image courtesy: Blockchain.info [16])

We treat these IDs as nodes and add directed edges from the nodes in the Input field to the nodes in the Output field. Thus the corresponding subgraphs in the Transaction Network and Public Key Network would be Figure 1.2a and Figure 1.2b.



(a) Transaction Subgraph



(b) Public Key Subgraph

Figure 1.2: Example Subgraphs

We also apply the aforementioned Multi-Input Heuristic and find the corresponding User Network. After application of the heuristic, Pk1 and Pk2 contract to U1 and Pk3 becomes U2. The U1 node should ideally have two self loops but to keep things simple, we consider a single self loop in our study. Figure 1.3 shows the scenario.

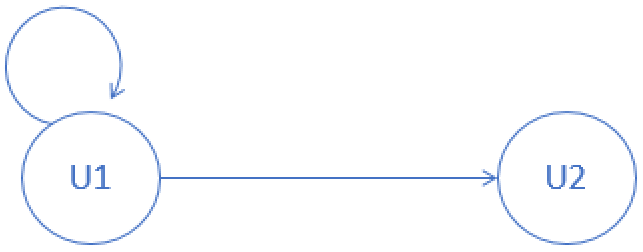


Figure 1.3: Example User Subgraph

1.1.1 Example Construction

Consider a set of transactions as depicted in Figure 1.4.

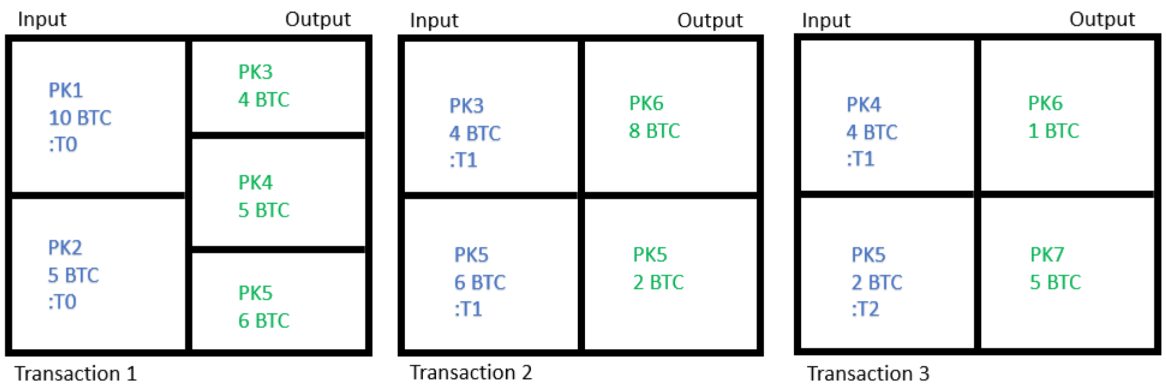


Figure 1.4: Example Set of Transactions (T0 is a fictional transaction included to maintain consistency)

Given this information, the following networks can be made,

Transaction (Tx) Network

Since Tx2 invokes Tx1 in its input line and Tx3 invokes both Tx1 and Tx2, we get Figure 1.5 as the resulting Transaction Network.

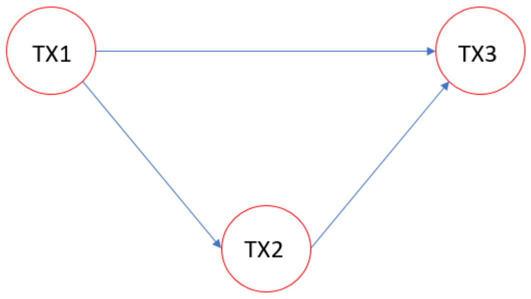


Figure 1.5: Example Transaction Network

Public Key (PK) Network

Figure 1.6 is the resulting Public Key Network. Tx1, being a 2-input-3-ouput Tx results in the 6 edges existing between PK1, PK2, PK3, PK4 and PK5. There exists a self loop on PK5 because it gives 2 BTC back to itself in Tx2 and there are multiple edges between PK5 and PK6 because PK5 sends money to PK6 twice, both in Tx2 and Tx3.

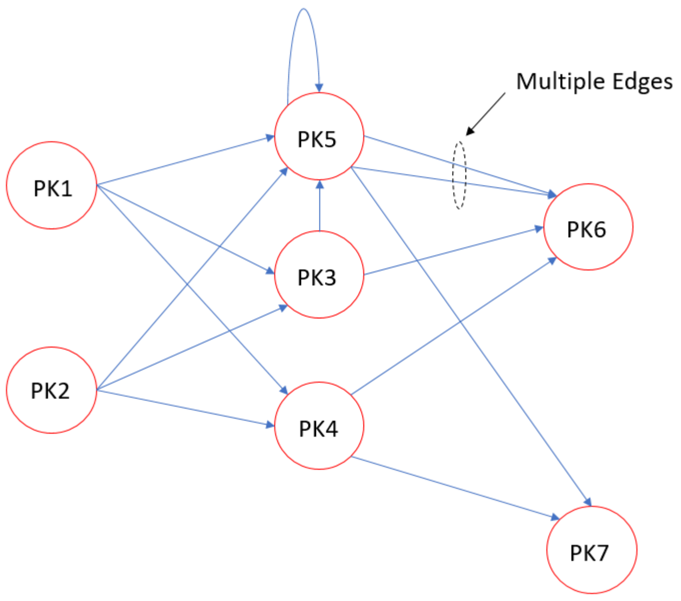


Figure 1.6: Example Public Key Network

User Network

Figure 1.7 is the resulting user network. This user network is made by using Multi Input Heuristic. PK1 and PK2 appear together in the input of TX1 and hence are mapped to a single user U1. Similarly PK3, PK4, and PK5 belong to input of Tx2 and T3 pairwise and hence are mapped to a single user U2. PK6 and PK7 are mapped to users U3 and U4 respectively. The multiple edges of the Public Key network are appropriately re-wired in this structure.

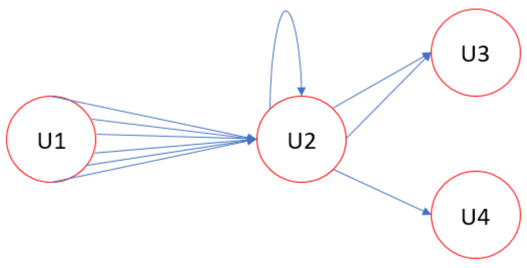


Figure 1.7: Example User Network

Public Key Social Network

Figure 1.8 is the resulting social representation of the Public Key Network (Figure 1.6). The edges become undirected and all multiple edges are merged into one edge.

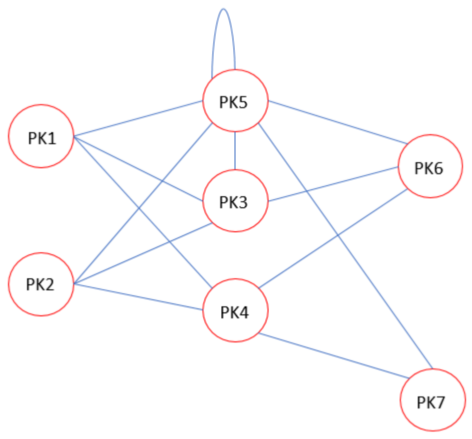


Figure 1.8: Example Public Key Social Network

User Social Network

Figure 1.9 is the resulting social network interpretation of the original user network. Once again, the multiple edges are merged into one single edge, and all edges become undirected.

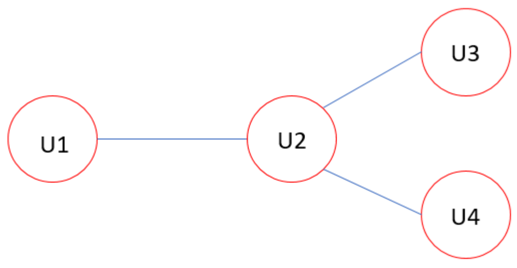


Figure 1.9: Example User Social Network

Public Key Money Network

Figure 1.10 is the resulting public key money network resulting from the example set of transactions. Here each edge not only points the direction of transfer but also contains the amount being transferred.

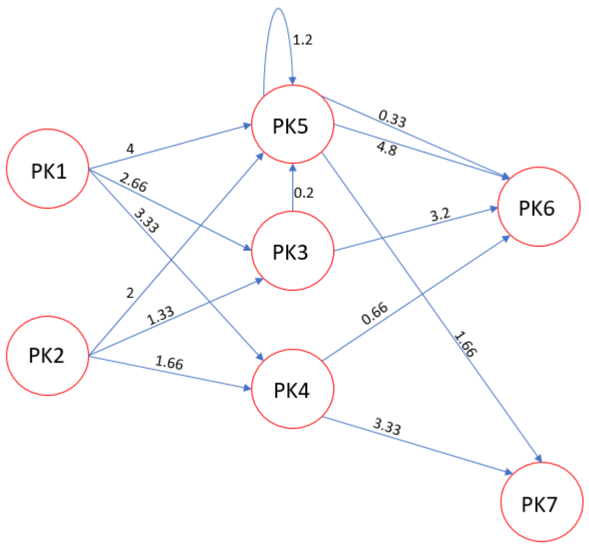


Figure 1.10: Example Public Key Money Network

For calculation of edge weights we use the following method,
For a m-input-n-output transaction, let $x_1, x_2, ...x_m$ be the input amounts inputted by public keys $IPK_1, IPK_2, ...IPK_m$ respectively. Let the output amounts be $y_1, y_2, ...y_n$ being given out to public keys $OPK_1, OPK_2, ...OPK_n$. Then, the edge weight between IPK_i and OPK_j is

$$weight(e_{ij}) = \frac{x_i y_j}{\sum_{k=1}^{n} y_k}$$

For example, the edge between PK1 and PK5 gets

$$\frac{10 \times 6}{4 + 5 + 6} = 4$$

as the weight. Other weights are shown in the Figure 1.10.

Chapter 2

Network Properties of Bitcoin

We gathered data from block 200,000 to 205,000, that is, 5,000 blocks and computed various statistics over it. These blocks contain data accumulated from 22nd September 2012 to 26th October 2012 i.e., five weeks of data. Our Public Key (PK) Network has 1,224,196 vertices (PKs) and 5,015,244 edges. In comparison, [RS13] attempts to quantitatively analyse the Bitcoin network too. Their graph contained the first 180,000 blocks of the network and had 3,120,948 PKs. Note how our graph spans only 5,000 blocks but still is 1/3rd the size of their graph. This is due to the huge increase in the popularity of Bitcoin beginning from that time period. Note also, that we are treating Bitcoin as a Complex Network and trying to study its network theoretic properties and not its anonymity/cryptographic aspects.

2.1 Degree distribution

A node in a directed network has two different degrees, the in-degree, which is the number of incoming edges, and the out-degree, which is the number of outgoing edges. The degree distribution $P(k)$ of a network is then defined to be the fraction of nodes in the network with degree k. Thus if there are n nodes in total in a network and n_k of them have degree k, we have

$$P(k) = n_k/n$$

On similar lines, the in(or out)-degree distribution $P_{in}(k)$ of a network is then defined to be the fraction of nodes in the network with in(out)-degree k.

$$P_{in(out)}(k) = n_k/n$$

Most networks in the real world have right-skewed degree distributions, meaning that a large majority of nodes have low degree but a small number, called "hubs", have high degree. Some networks, notably the Internet, the world wide web, and some social networks are found to have degree distributions that approximately follow a power law: $P(k) \sim k^{-\nu}$, where ν is a constant. Such networks are called scale-free networks and have attracted particular attention for their structural and dynamical properties. [Wik16b]

2.1.1 In-degree distribution

Figure 2.1 shows the log-log plot of in-degree frequency versus unique in-degree values. It shows the characteristic "line with negative slope" which might lead someone to conjecture that the network follows power-law distribution. But authors of [RH13] note that it is not the case. In fact, the network does not follow preferential attachment [Sim55] as the new PKs stick to the PKs from which they derive the money. Also, the average degree of the graph is 8.2 and only 4.06% of the nodes have in-degree $\geq$ the average degree. But in a scale free network, nodes with $\geq$ average degree occur relatively commonly.

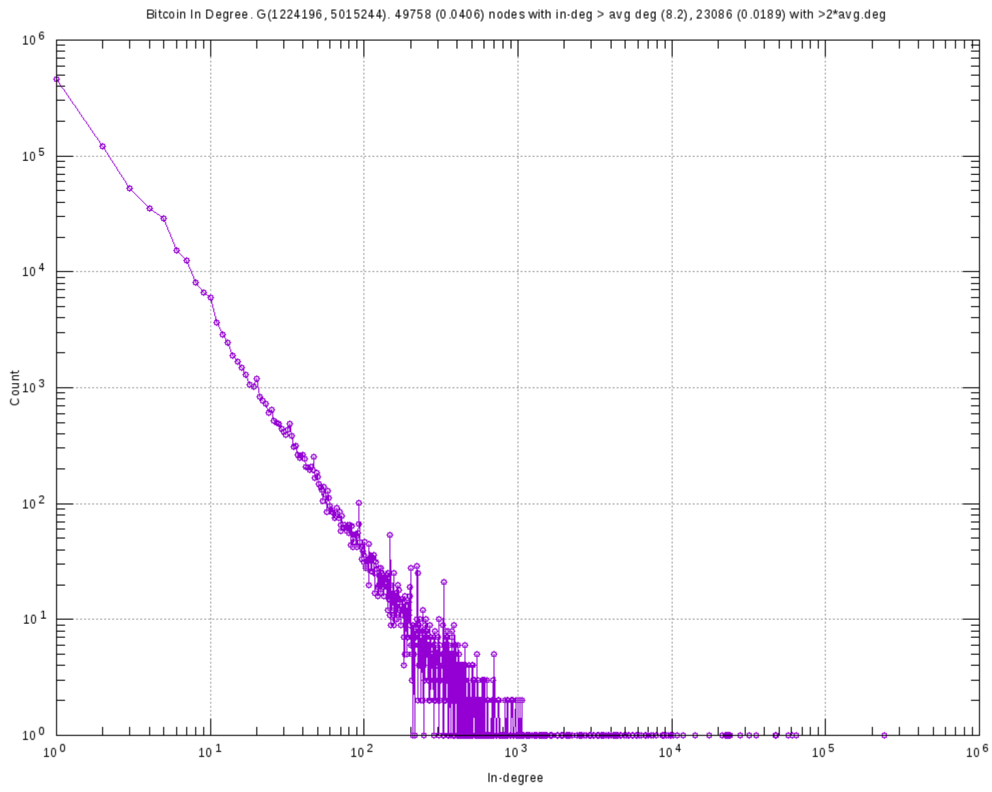


Figure 2.1: In-Degree Distribution of the PK Network. Blocks 200,000 to 205,000

2.1.2 Out-degree distribution

Figure 2.2 shows the log-log plot of out-Deg frequency versus unique out-Deg values. Notice how the nodes with out deg 2,4,6.. are relatively higher in number than their odd valued neighbours. This is not by accident.
The relative abundance of PKs correspond to the relative abundance of Tx with 2 output entries. In Bitcoin scheme, the "change" is not handled automatically. Bitcoin wallets make the payment using a previous Tx and then they transfer the remaining amount, the "change", to a newly generated Public Key causing the abundance of nodes with out-degree 2. Now, if the same PK is involved in multiple Tx, then the out-degree increases as 4,6.... In contrast, Tx with 1 output occurs with much less frequency (1/10). Nodes of degree 3 form when Tx with 3 outputs occur or if a PK appears in the input of two Tx containing 1 and 2 outputs. This explanation can be extended to apply to the zig-zag pattern observed at low out-degree values.

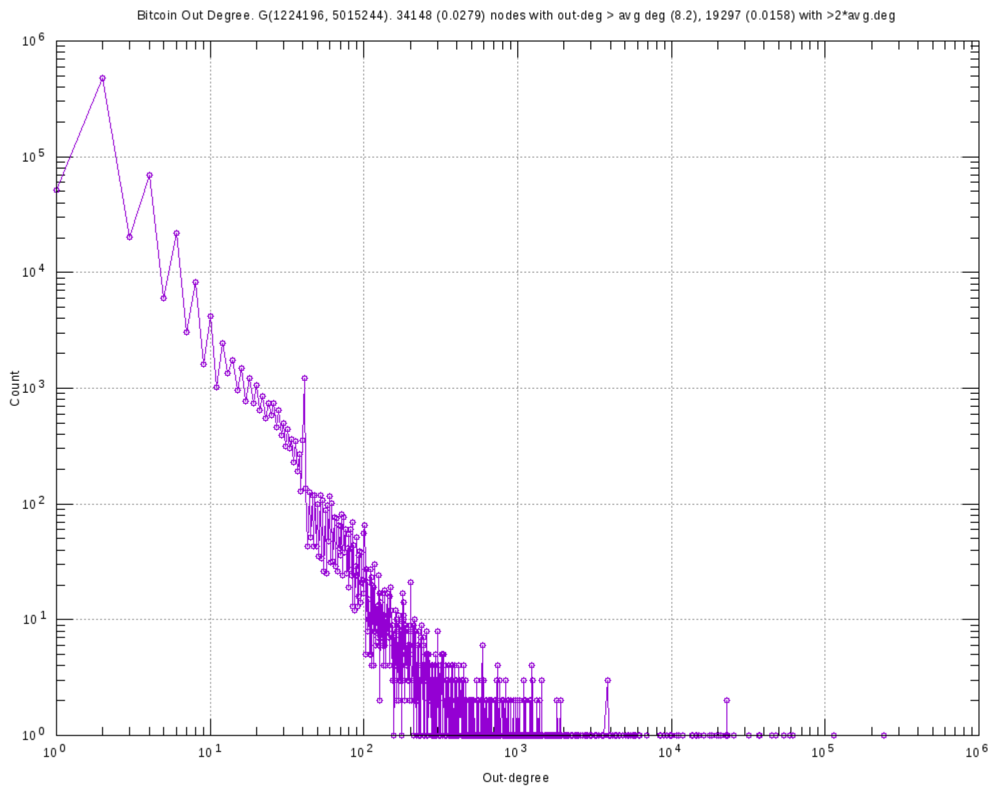


Figure 2.2: Out-Degree Distribution of the PK Network. Blocks 200,000 to 205,000

2.2 Clustering Coefficients

The local clustering coefficient of a vertex (node) in a graph quantifies how close its neighbours are to being a clique (complete graph). Duncan J. Watts and Steven Strogatz introduced the measure in 1998 to determine whether a graph is a small-world network. A graph $G = (V, E)$ formally consists of a set of vertices V and a set of edges E between them. An edge e_{ij} connects vertex v_i with vertex v_j. The neighbourhood N_i for a vertex v_i is defined as its immediately connected neighbours as follows:

$$N_i = \{v_j : e_{ij} \in E \lor e_{ji} \in E\}$$

. We define k_i as the number of vertices, $|N_i|$, in the neighbourhood, N_i, of a vertex. The local clustering coefficient C_i for a vertex v_i is then given by the number of links between the vertices within its neighbourhood divided by the number of links that could possibly exist between them. For a directed graph, e_{ij} is distinct from e_{ji}, and therefore for each neighbourhood N_i there are $k_i(k_i - 1)$ links that could exist among the vertices within the neighbourhood (k_i is the number of neighbours of a vertex). Thus, the local clustering coefficient for directed graphs is given as [WS98]

$$C_i = \frac{|\{e_{jk} : v_j, v_k \in N_i, e_{jk} \in E\}|}{k_i(k_i - 1)}$$

As an alternative to the global clustering coefficient, the overall level of clustering in a network is measured by Watts and Strogatz[WS98] as the average of the local clustering coefficients of all the vertices n:[And09]

$$\bar{C} = \frac{1}{n}\sum_{i=1}^{n} C_i$$

A graph is considered small-world, if its average local clustering coefficient $\bar{C}$ is significantly higher than a random graph constructed on the same vertex set, and if the graph has approximately the same mean-shortest path length as its corresponding random graph.[Wik16a]
The number of open triads are the number of unique (upto edges) open walks of length 3 (or P_3) and the number of closed triads are the number of unique (upto edges) triangles (or K_3). The sense of direction is ignored but multiple edges are counted individually while counting these walks.
The clustering coefficients, both global and local, have been calculated in accordance with [WS98]. We calculate the measure in two ways.
1. **Graph**: We consider the graph of the PK Network. One self loop allowed per node and maximum one undirected edge between any two distinct vertices.
2. **Network**: We consider the PK Network as a directed multigraph. One

self loop allowed per node but any number of directed edges between any two distinct vertices.
We use the notation $Ne + M$ to denote $N \times 10^M$. (For example, $4e + 5 = 40,000$). Each time, we extract the Maximum Maximal Connected Component (MMCC) of the given graph (Maximum Maximal Strongly Connected Component (MMSCC) in the Network case) and calculate the clustering coefficients over this connected subgraph.
We also do something novel, we apply a Deanonymizing heuristic[Nic15][RH13][RS13], the Multi-Input Heuristic, to the PK network and convert it to the "User" Network.
The Multi-Input Heuristic works in the following way. If two Public Keys appear in the Input line of the same transaction then both the Public Keys are associated with the same user. We thus concatenate the two Public Keys belonging to a single user and rewire the connections in the graph/network accordingly.
Some other successful heuristics are mentioned in [Nic15] but they are prone to false positives and hence we shall restrict ourselves to the Multi-Input Heuristic.
Note that we are a little audacious in our use of the term "Users" and the "User network" because the Multi-Input Heuristic leaves out some PKs of a user and hence the number of "users" is still much more than the actual numbers of users in the system. But as [HF16] shows, the heuristic is fairly effective.

2.2.1 PK Graph Clustering Coefficients

Figure 2.3 shows the plot of the local clustering coefficient of nodes in the PK Graph. The MMCC of the PK graph contains 804,427 nodes (PKs). The number of open triads is 1,916e+6 while the number of closed triads is 3.6e+6.
The average clustering coefficient is 0.1572.

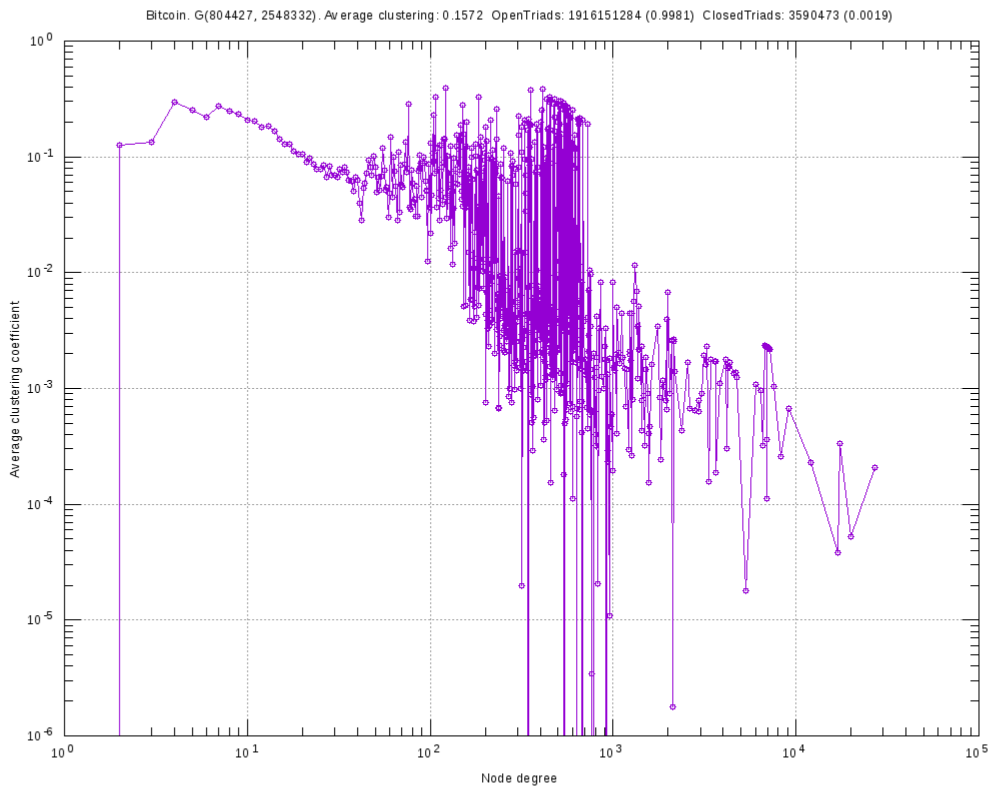


Figure 2.3: Local Clustering Coefficient Plot of the PK Graph. Blocks 200,000 to 205,000

2.2.2 Clustering Coefficients of the User Graph

Figure 2.4 shows the plot of the local clustering coefficient of nodes in the User Graph. The User Graph is obtained on application of the Multi-Input Heuristic on the PK Graph. The MMCC of the User Graph contains 456,622 nodes (Users) and almost half the number of edges than in the original PK Graph. The number of open triads decreases to 436e+6 while the number of closed triads becomes 0.9e+6. One could argue that this happens because of the decrease in the number of vertices and edges (as we are considering a Graph, and after concatenation of two vertices, multiple edges are lost). Figure 2.5 provide an example of the effect. The nodes in figure 2.5a enclosed inside the green circle are being concatenated. The resulting structure is 2.5b
The average clustering coefficient increases to 0.1869. We were considering the single undirected edge Graph structure to capture social network aspect of this network. The increase in the global clustering coefficient after application of the Multi-Input Heuristic shows that the network has become more "small-world" [WS98] than before.

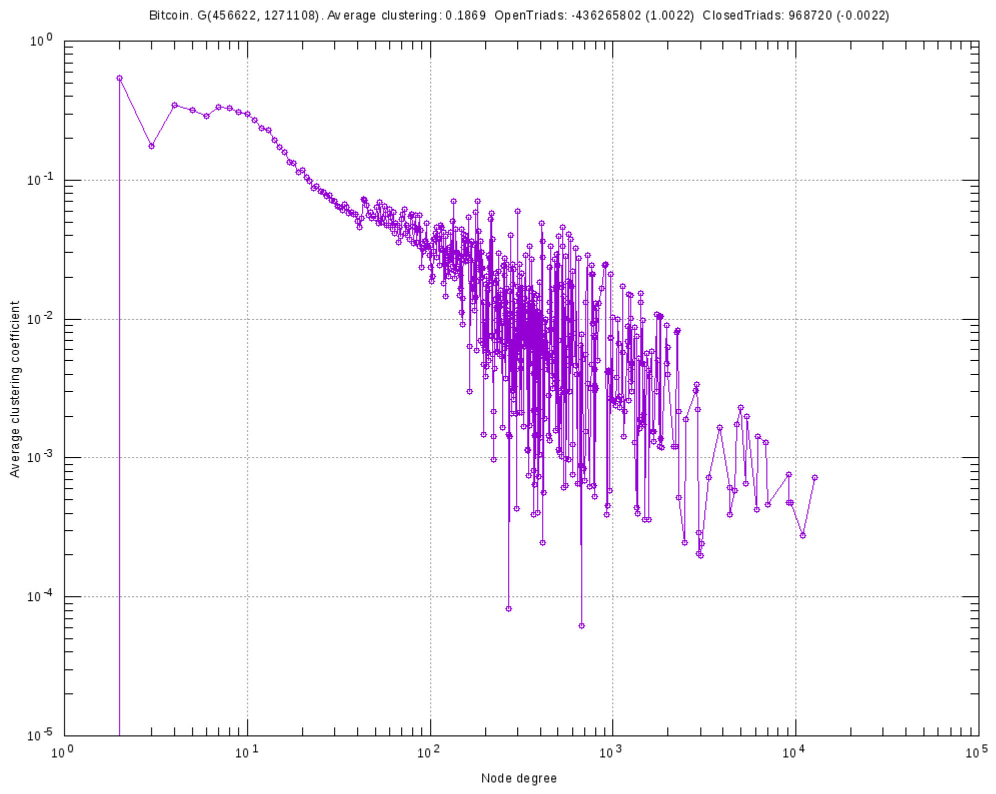
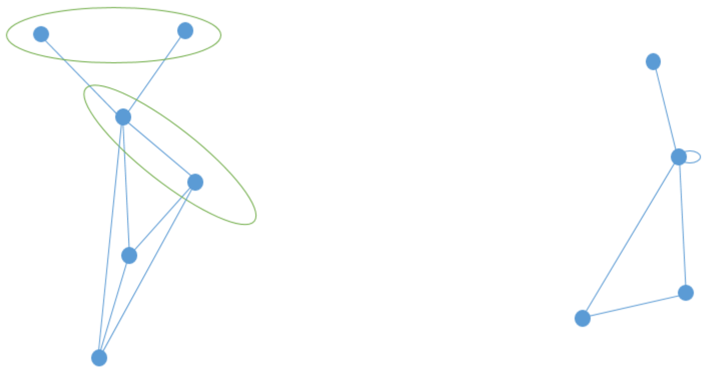


Figure 2.4: Local Clustering Coefficient Plot of the User Graph. Blocks 200,000 to 205,000



(a) Open Triads = 6
Closed Triads = 3

(b) Open Triads = 2
Closed Triads = 1

Figure 2.5: Example effect of Multi-Input Heuristic on PK Graph

2.2.3 PK Network Clustering Coefficients

Figure 2.6 shows the plot of the local clustering coefficient of nodes in the PK Network. The Maximum Maximal Strongly Connected Component (MMSCC) of the PK Network contains 483,771 nodes (PKs). The number of open triads is 1395e+6 while the number of closed triads is 13.0e+6

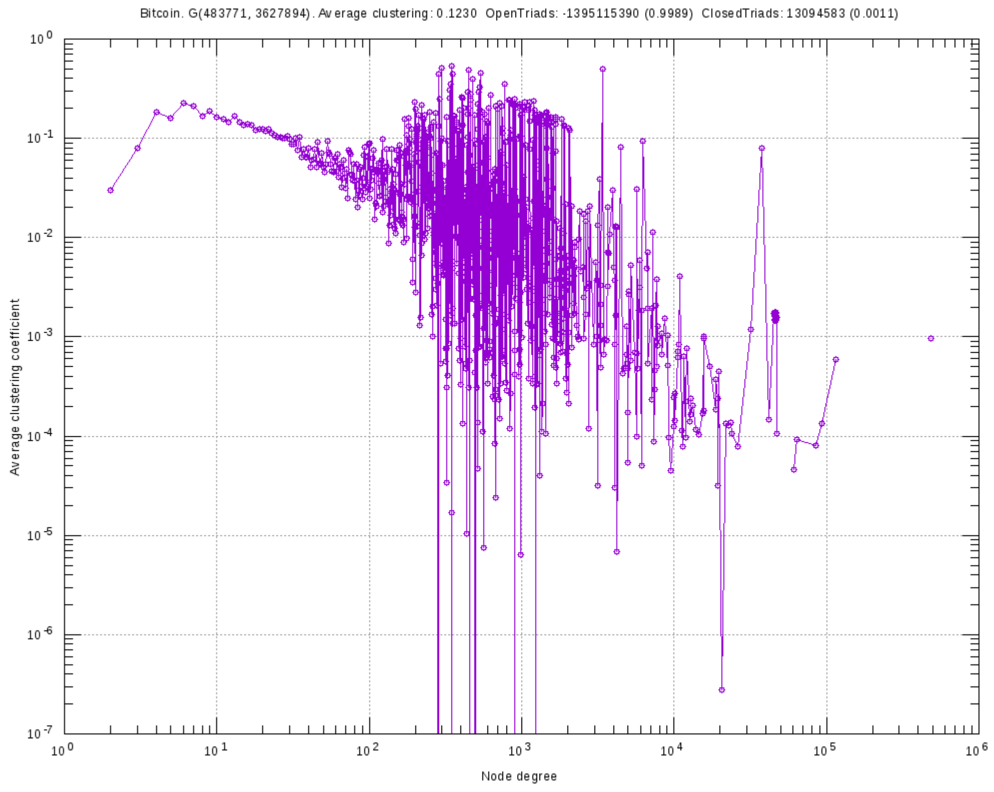


Figure 2.6: Local Clustering Coefficient Plot of the PK Network. Blocks 200,000 to 205,000

2.2.4 Clustering Coefficients of the User Network

Figure 2.7 shows the plot of the local clustering coefficient of nodes in the User Network. The User Network is obtained on application of the Multi-Input Heuristic on the PK Network. The MMSCC of the User Network contains 233,457 Users (again, almost half the original size) but almost the same number of edges as in the original PK Network. The number of open triads increases to 1887e+6 while the number of closed triads becomes 12.1e+6. This time, since we are allowing multiple edges, concatenating two vertices and rewiring connections causes an increase in the number of open triads passing through that concatenated node. Closed triads don't suffer much. Figure 2.8 provide an example of the effect. The nodes in figure 2.8a enclosed inside the green circle are being concatenated. The resulting structure is 2.8b. Note how the resulting structure has more open and closed triads than the previous one.

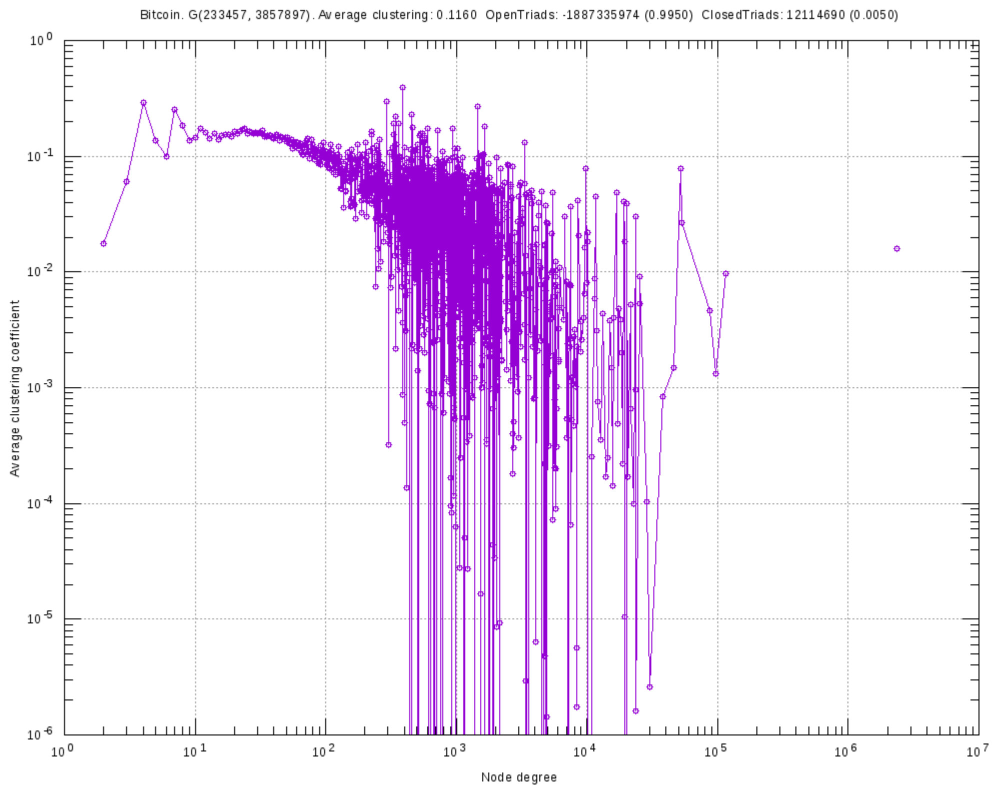
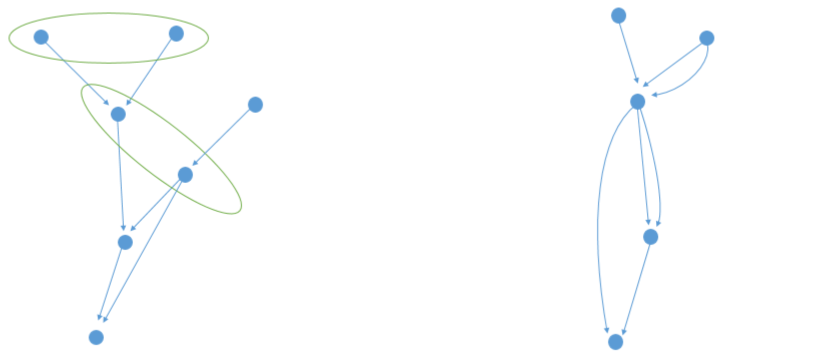


Figure 2.7: Local Clustering Coefficient Plot of the User Network. Blocks 200,000 to 205,000



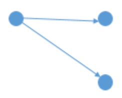
(a) Open Triads = 6
Closed Triads = 1

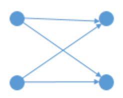
(b) Open Triads = 9
Closed Triads = 2

Figure 2.8: Example effect of Multi-Input Heuristic on PK Network

2.3 Motifs in the PK Network

Network motifs are sub-graphs that repeat themselves in a specific network or even among various networks. Each of these sub-graphs, defined by a particular pattern of interactions between vertices, may reflect a framework in which particular functions are achieved efficiently. Indeed, motifs are of notable importance largely because they may reflect functional properties. They have recently gathered much attention as a useful concept to uncover structural design principles of complex networks[MSK12]. Although network motifs may provide a deep insight into the network's functional abilities, their detection is computationally challenging.[Wik16c]
The PK Network is highly connected and has lots of multiple edges and self loops. To identify the primary motifs in the network we look at the Tx network of the data. Note that a n-input-m-output transaction creates the complete bipartite graph $K_{n,m}$ whose all edges are directed from the Public Keys appearing in the output line to the Public Keys appearing in the output line. We see that the most common motifs are 2.9aand 2.9b occupying almost 60% and 14% of the total volume. They are formed on a 1-Input-2-Output and on a 2-Input-2-Output Transactions respectively. One more common motif is 2.9c which is formed when one of the input PK is used in the output line of the Tx.


(a) 1-Input-2-Output Tx


(b) 2-Input-2-Output Tx

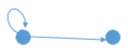

(c) 1-Input-2-Output Tx with a PK common in Input and Output.

Figure 2.9: Notifs occuring in the PK network due to corresponding type of Transactions in the Tx network.

Chapter 3

Finding Interesting Nodes in the Bitcoin Network

The problem of finding important nodes in any given complex network, be it financial, social, a network of hyperlinks (web pages) or Bitcoin is an active area of research. There have been some attempts in this direction for Bitcoin too. [FKP15] applied the traditional Google's PageRank[Pag+99] to the Bitcoin data of one day and showed that it indeed was able to highlight important nodes in the graph. [Mar15] also attempts to deliver something similar and uses a Pagerank model along with transaction weights and sum them to yield an importance score to rank users in the network.
In similar spirit we also ask the question of finding the most important users in the network and give a Weighted Pagerank model, called BitRank, to do the same.
The effectiveness of Pagerank lies in the fact that it uses the intrinsic network structure to predict the importance of a vertex. If we are able to correctly define a markov process and use Pagerank "in the right sense", we find that it can be used in quite diverse applications. [Gle15] provides a number of examples in this respect including Pagerank's applications to movie recommendation systems, Protein docking problem and of course to search engines operating on the classic web structure. *(the author would encourage the reader to have a look)*

3.1 The Public-Key-Money Network

Recall section 1.1.1 and the construction of the Public-Key-Money-Network from a set of transactions. For this analysis we took sets containing thousands of interconnected transactions and made the corresponding network. Each edge in this network represents the total flow of Bitcoin happening from the tail of the edge to the head of the edge in the subgraph we have considered. For calculation of edge weights we use the following method, For a m-input-n-output transaction, let $x_1, x_2, ...x_m$ be the input amounts inputted by public keys $IPK_1, IPK_2, ...IPK_m$ respectively. Let the output amounts be $y_1, y_2, ...y_n$ being given out to public keys $OPK_1, OPK_2, ...OPK_n$. Then, the edge weight between IPK_i and OPK_j is

$$weight(e_{ij}) = \frac{x_i y_j}{\sum_{k=1}^{n} y_k}$$

Now after getting the edge weights between every pair of PKs (vertices), note that multiple edges my still be present with their own unique weights between these PKs. Hence before calculating BitRank, we sum these different weights and set their cumulative weight as the weight of that edge. Mathematically,

$$Cumulative\_Weight(e_{ij}) = \sum_{k=1}^{k=n} e_{ij}^{k}$$

Note that this cumulative weight is nothing but the total amount being transferred from IPK_i to OPK_j in all the transactions present in the subset of transactions that we have considered.

3.2 BitRank

We now present the main result of this chapter, the BitRank algorithm. But before delving into its details, let us brush up on the Pagerank problem and its solution.

3.2.1 The PageRank Problem

As pointed out by David F. Gleich[Gle15], there are many slight variations on the PageRank problem, yet there is a core definition that applies to almost all of them, which arises from a generalization of the random surfer idea. Pages where the random surfer is likely to appear have large values in the stationary distribution of a Markov chain that, with probability α, randomly transitions according to the link structure of the web, and with probability $1 - \alpha$ teleports according to a teleportation distribution vector v, where v is usually a uniform distribution over all pages. In the generalization, we replace the notion of "transitioning according to the link structure of the web" with "transitioning according to a stochastic matrix P." This simple change allows us to to use any stochastic matrix P and gives a method to rank its nodes on basis of the incoming transition probabilities. Let P_{ij} be the probability of transitioning from page j to page i (or, more generally, from "thing j" to "thing i"). The stationary distribution of the

PageRank Markov chain is called the PageRank vector x, which is the solution of the eigenvalue problem

$$(\alpha P + (1-\alpha)ve^T)x = x. \tag{3.1}$$

where e is an all 1's vector. There are many standard ways to go on about solving the PageRank problem. The one which we have used is the power method corresponding to the simple iteration,

$$x_{k+1} = \alpha P x_k + (1-\alpha)v \tag{3.2}$$

where x_0 is either the zero vector or $x_0 = v$

3.2.2 The BitRank Algorithm

Algorithm 1: The BitRank Algorithm

```
Result: Public Keys ranked on the basis of their importance
Input : The Public-Key Money Network as a sparse matrix $A_{n\times n}$
$d1 = \{\}$ //d1 is a hash table with Key=UniqueId and
  Value=Cumulative_Weight as its (Key,Value) pair
$\epsilon = 0.001$
$maxIter = 20$
for $i,j$ from 1 to $n$ do
    $UniqueId = n \times i + j$
    if $UniqueId$ not in $d1$ then
        $d1[UniqueId] = 0$
        $d1[UniqueId] = d1[UniqueId] + A_{i,j}$
    else
        $d1[UniqueId] = d1[UniqueId] + A_{i,j}$
    end
end
for $i,j$ from 1 to $n$ do
    $M_{i,j} = d1[UniqueId]$
end
for $i$ from 1 to $n$ do
    $rowSum[i] = \sum_{j=1}^{j=n} M_{i,j}$
    for $j$ from 1 to $n$ do
        if $rowSum[i]$ not equal to 0 then
            $Sink[i] = 0$
            $M_{i,j} = M_{i,j}/rowSum[i]$
        else
            $Sink_i = 1/n$
        end
    end
end
$Teleport = (1/n, 1/n, 1/n....1/n)_{n\times 1}$
$iterations = 0$
while $(\|x_k - x_{k-1}\| < \epsilon$ and $iterations < maxIter)$ do
    for $j$ from 1 to $n$ do
        $V = M_{:,j}$ //$M_{:,j}$ is the $j^{th}$ column vector of M
        $x_{k+1} = x_k \cdot (\alpha(V + Sink) + (1-\alpha)Teleport)$
    end
    iterations=iterations+1
end
Output: x
```

Where,

- A is the input Public-Key Network in the sparse matrix format. The format we follow is the (row,column,value) triple for the non-zero entries.
- M is a row-stochastic matrix which we construct from A by first using Cumulative edge weights of the edges in A *(line 14)* and then by normalizing it by the row sums *(line 21)*
- Sink is the vector which recognizes the sinks in the network (nodes with zero out-degree) *(line 20 and 23)*. We have assumed that a random surfer can go out via this node to any other node uniformly. Hence in BitRank, we populate the entries corresponding to the sinks with the uniform distribution vector $(1/n, 1/n, 1/n....1/n)^T$.
- Teleport is the uniform distribution vector here. It is the parameter which can also be used to focus on a set of nodes instead of all of them uniformly. The way to do that is to assign higher values to the nodes of interest and zero to others along with the constraint $\sum_{i=1}^{i=n} Teleport_i = 1$, that is the sum of all entries in the Teleport vector should equal 1.
- UniqueId is a unique number assigned to each edge. The formula *(line 5)* works because the indexing in our matrix is from 1 to n. We use these UniqueIds to give cumulative weights to all the multiple edges in the network and reduce them to single edges.
- d1 is a hash table storing UniqueIds and their respective cumulative weights. Hash tables provide a fast way to search and we have used them throughout our codes to speed them up.

3.3 Concluding remarks and Examples

BitRank produces mildly different results than simple Pagerank when applied to the Bitcoin network. This difference, although subtle may prove vital when considered in data mining and analysis operations on the network. One persistent problem in generic algorithms is of false positives being included in the results. We will now present examples and show our algorithm performs better in exactly this regard. Throughout our discussion below, SP denotes Simple Pagerank and BR denotes BitRank

3.3.1 Example set 1 *(Blocks 400,000 to 400,003. 4536 Nodes)*

1. Public Key: 17h5rSzFFm9c3L9ce1vSxvpBweSQ6LgzLr
   SP = 7 BR = 3251
   Included in the output of the transaction
   70e36afe83a1adb86b193a64f00fb3fe7db8c0720f4516bc4c88f11f6d8e2b6e, this public key gets 0.01% (0.01 BTC) of the money being transferred in the transaction. The other amount , which equals 102 BTC is definitely a sort of payment in lieu of goods or a neat amount stored as a deposit. In this case, BitRank divides the importance of two public keys in ratio of their incoming amounts and correctly deprecates the rank of this node to 3251 while simple Pagerank still ranks this as the 7th most important node of the set.

2. Public Key: 1MDzUDQwBUdhSoyv8MHoH4DdvbA7UnF2fJ
   SP = 9 BR = 3627
   Included in the output of the transaction
   f1dd43e9a1283c433ad74b0e3c45bc69d4239ded26db708d43e52e177fcb68a5, this public key receives 0.001% (again, almost 0.01 BTC) of the money in the transaction. With exact same analysis as the previous example, BitRank again correctly deprecates its rank in the set.

3. Public Keys:
   (A): 1C3dZ31vPHbV3G3HRecVe9KLZYUqBTAVpn and
   (B): 13czSzu48Qn9Y1yohjsLeap28rnnKg1WgD
   SP = 15, 66 BR = 448, 274
   Included commonly in the output of the transaction
   a1e92c89ad77749a1c55680dc03b48b76f829f114134d8dcc891daef3763bbe5, these public keys get reverse ranks by the two algorithms.
   This is happening because the Public Key (A) appears in 4 transactions in its lifetime while (B) appears in only 2, given that those transactions were of similar type, (A) gets a higher Pagerank than (B) purely because of a higher in-degree.
   But (B) gets a higher share of money in both its transactions while (A) consistently receives the smaller share of the transaction. In the Transaction where they appear together, (B) receives 1000 times more money than (A).
   This makes the rank of (B) better than (A) in BitRank.

4. Public Key: 37xHeigAr8ThwvfNv3LjL1A6s7LUhqA3JC
   SP = 435, BR = 9
   We suspect that this Public Key belongs to a user and not a web service operating in the BitCoin network. The reason for this belief

are many fold. Firstly, this public key transacts infrequently and in small amounts (as compared to companies in the Bitcoin domain). Secondly, this public key repeatedly deposits the change back to itself (in 103 out of 105 transactions), a telltale sign of a casual Bitcoin user. Nodes like this get a high BitRank because the change returned to oneself is usually higher than the payment made which forces the weight of the self-edge to be greater than all other edges combined. In contrast, simple Pagerank gives equal importance to the self-edge and other edges, making public keys like this deprecate in rank.
This behaviour is seen in many instances and it is safe to say that highlighting nodes which can possibly be humans is a boasting feature of BitRank.

5. Public Key: 1KB4dzL63wJBqehUMFmSLA1dwsC7yUGaGY
   SP = 310, BR = 13
   Again, we suspect that this Public Key belongs to a casual user in the network. With the same analysis as the previous example, this Public Key sends change to itself in 4 out of 6 transactions with a greater than 0.5 weight. Consequently, it gets a higher BitRank but a low PageRank.

6. Public Key: 197NdraAGmPqnN5QLxLBf1u7JA649E8q6Z
   SP = 162, BR = 17
   Included in the transaction
   bafb1270ca2df875396fe7b4a5a2c418d7e44b06e7de8265b5fb2a5acec07604
   this Public Key receives the highest amount amongst the 21 Public Keys receiving money from that transaction. Consequently BitRank enumerates it as the highest amongst those and promotes its rank much more than simple PageRank.

3.3.2 Concluding remarks

Motivated by the previous examples we computed BitRank and Pagerank for more and bigger datasets and were able to conclude that the set of Public Keys which gets affected significantly in their ranking *(≥100 for example)* can be divided into two distinct classes.

Class 1:

Public keys which get a high BitRank but low Pagerank are the ones which awfully mimic the behaviour of a casual user. Indicating that it might very well be one. The ability of our algorithm to dilute importance in ratio of money proves beneficial when considering a normal user of bitcoin network.

Suppose a person has one Bitcoin and he makes a payment of 0.3 Bitcoin returning 0.7 Bitcoin of ”change” to itself. Now suppose he again indulges in a transaction and makes a payment of 0.3 Bitcoin returning the 0.4 Bitcoin ”change” to itself. In pagerank matrix, this information would result in equal importance being diluted from the user making the payment to itself and the website/shop to which the payment is made to. Now these shops is anyways receiving payment from many customers and hence would get a high pagerank by accumulating all these small shares of importance. But consequently, a casual user gets hidden amongst all the other high ranked nodes. Bitrank overcomes this by returning back the importance of a user just as he is returning the ”change” to itself, that too in ratio of the payments made. Multiple ”changes” result in a ”feedback” effect promoting the rank of the casual user in the network considerably.

Class 2:

Public Keys which get a high Pagerank but significantly low BitRank. These Public keys are usually included for returning the change of the transaction or as a micro payment in the transaction. Pagerank, by its nature promotes its rank to its neighbours rank in that transaction. Bitrank on the other hand, gives it only as much importance as the fraction of money it receives. In this respect BitRank works by removing false positives being outputted by the simple Pagerank.

3.3.3 Example set 2

(Blocks 401,000 to 401,010. 18,268 Nodes)

We examine the top 100 nodes of each set, that is top 100 ranked by Pagerank and top 100 by BitRank and analyse the places where they differ significantly *(difference in rank $\geq$ 500)*.
With the definition of the aforementioned classes in our hand,

Public Keys falling in Class 1

Number of Public Keys = 15

- 16VtBHDd8BksWSP3kZnmwPcVirBfkjGS4R
- 19Jnx1knT1jnsvk17qsGLL8p1ZLHEsG1s5
- 1L9PxYYg9MBT2x9dDXMjmkzE2WEvdksz9b
- 1NPRFZnutc2WVmQt5iPmpm5LTYSj4Axc49
- 1QF4LDLWBaCvDzUVEBUXeUTDvK4acgJzZD

- 1Dkr4LAhSwHDfgG69y69ApYNsQNrVe4Vty
- 1PPiaDqkLEQBmKxzF1gmSrCAbCsJTDfKMu
- 1HD9CoV3nGXY2qxNBumvqQp1VxSLJ3ucEZ
- 1snwoofRcNMC497TezM2QqRPQhwK4yJJq
- 1KXXw4GY5CHwmnJNV5K8zCoiJx8mHp4Eah
- 19GSubJ5mcTGKFHQcmg8ga72P4ywToEzJQ
- 13j1NxQShNfdXaziYkGara5E5dMDER2VC7
- 1Jc3ipkYBLTKi2NB6qpu85b5nhkWED32mh
- 128WmVJiwHEu8wyqgsx4A7p7pLKmTiVBYS
- 1Dgmh2ffv3tsrEeRcXpRD7HYZQqWw5tpfa

Public Keys falling in Class 2

Number of Public Keys = 22

- 12XxjhbjB6k3MNdPeg6RsNhbp87MebVc9T
- 1LR4ymrbRWoeUMMBiRkUCmpmsuzi4DwDfs
- 1DMUcfqN4We4AUsoodxSZ7LrnPv4WTS3bD
- 1AavUZPq5K5vZynGPyEPwHLLbZYXdo9g8V
- 1MoDfXgRWsWTpQbP8Rb9EhcTftB93MV2kU
- 1KsJCVAcHSopxDNjqdYQfTXGz53YMafzqu
- 12hUULnBEDoLERGu8ctM9PRGGAi8YjwBFH
- 19RaPApdrwyND2rV5XvXC4mYzDBPbRuFB
- 1Bu8f2B5JHUrZBN93pBuXUAiexddjPnXPN
- 1E4Woii3Ef3XWShLxeAh9xRNgdLVWcMiND
- 1GJvZ8nmPp8BxYnb5n98iB5W91yUcoH2z4
- 17s4gEfM2LasPxDwZBJogQK4CmBn8PDVdY
- 1DUb2YYbQA1jjaNYzVXLZ7ZioEhLXtbUru
- 1JAwUkw4h3aLF8EHh2NA6RnPThz12nJRDt

- 1Po1oWkD2LmodfkBYiAktwh76vkF93LKnh
- 1bonesHWYzgDUFWAWTKqiV7BggLXSdJXj
- 17Dk6dVxcYTWS8UyFZp276Peegzfinej1G
- 1Ky6M61GpBcQ4EqSTtDm61cCzySa9TbAnQ
- 1HFgq6vnpwJxvaDsz8HGA2TgjYYU8HtyBQ
- 1PkpeCvrtDDm4ZcebWmAGCZeJJeNCg8nEU
- 18AqmQ8m7qofmVHF2C67Bs3bixCirqfgAC
- 1MruarwJGJzwni2MsL83G5YSkCcEVngUAj

3.3.4 Example set 3

(Blocks 402,000 to 402,020. 34,964 Nodes)

Public Keys falling in Class 1

Number of Public Keys = 25

- 1z1CGThCbyWiWFcBue9oWZD2NecyKBY6s
- 1LjCngBrb12CDBev7562Mxorjr1MZ8nRUQ
- 1Q7wmGwfmBzxiLhnPAtBcpS5CPb3QzCMcx
- 1HNvz5NsHBE5Cc8a7GiCNyx7PwHGtvQgU9
- 13RogPhRwfbBijDbq8iyNYGSS7yZfgPicC
- 1ATATT6Cn3uxfZFdfLmTqjgCP3TyhRSFKs
- 3D9Gi4JsBaFRfyyDHiC3QM4o3zreV6oD4G
- 1NMbgz89gBK5vofYXyMcK1msuEogA5x2Yd
- 13vmayHG3cU5JBZbXqi6vP6yh5chumcxBv
- 1PMii38r8i1Nk2GjtPX1JcJrXR81Xw6tCZ
- 13HawdQqWAtJRt8j5TX449GnNCTsncAM9Y
- 3EmtSeEEJEGaX7JKpmuupf5N2F4pQ7vDuM
- 3M1XKJMYxeKbdmLgwWWi7TPNtSuVKqEsJN

- 3FUcTQa2XWHfnCZQmUWEU18xQQeMuMuVgJ
- 124Z2Tur9SfjbAkmrSsvAkx2xdCLVbjCd1
- 1GngBG9m6SqEgnshv1Mf6TK6b4Ddz9RMQQ
- 39oM6nD6SrTashAJnud8iLfSEgRD8mgeWF
- 1HPoBXM7AR6xM4pSEnrDERvya7ojhuAJTF
- 13JDM6LAcEYNCprEzNJwh2zm9VH4kLrSJZ
- 18Ca1exLWFU5ij6pkXnXKHiMsFqDfUb81s
- 1Ds8oHhcQTUcbi2ptCAu8UWuiqkyWMkf52
- 1MuGtJai2ym1uHMYA2WTxDdoscitSJEamr
- 37vQmmLfThGCumry9nAapvtdbYc9gVmaa5
- 15gUwBcMh6pxRqty5HZpTZKhKzY6P7Ppez
- 1LEAkhk9Q7kTdck2p6oc5Vu3Loirinnyfk

Public Keys falling in Class 2

Number of Public Keys = 26

- 19hsC6gRbgDix6hncwRwNKQLPuugHuyBRk
- 1KZqmpLnJCfVhZ4VdFWj9Na5pJxk2sW6Y6
- 182F7t1KFtXghdxzun9SfyPc45xPfH6kXc
- 1Js4bsFqd4wffKJzg4iqAUZar7bHgc8Qd1
- 1JKhSdd4xqX1QKkQKAtpdYXevcRxjwo648
- 1Hbe7HhwgrXm4nAmB6x1BszY1TSywQu8Vo
- 1MPxhNkSzeTNTHSZAibMaS8HS1esmUL1ne
- 1H9mTvAWRgptipBrnUxp7eo5VuFzP7Bm4G
- 3DLZn6M5zS9ZX6VLqGgAx52cV1kwSWUQaA
- 3QpMFVNY2qVgsqKA8E88XBECVrPZy9xjoz
- 1AavUZPq5K5vZynGPyEPwHLLbZYXdo9g8V
- 1Bav4RPACQBARTCkEJMczirbpdnhJEkrRq

- 1PatFGA4NJU5fHA2vaxPSb4cXNK6jWNpaY
- 17c9BLokA14tJStxMsHsUvpGbP9oVBbd3Q
- 1Aq8x1aLQWR6jqgNXNUgcJ8HMbNLCXa3VB
- 1AzDMT5XY8CbHWsEP8444rcYKSSfkteUXW
- 1NvcXkqBCYqPYeMZXxrm3LDg9ZpZMtSTLr
- 1B52rhXkVgL4tBfh5NzY4nqGp8XfjE3fSC
- 1Li4MGEfvU3fJQQqU6UBqq1LbvfCRuJ4J1
- 16cyzcPKMsxKsKRLBq8Z7RmMkqqeSUjeB
- 1HQyD8iTmi3vFTFzVEckKVWPVA9KmcV5sj
- 196w7QFU19Brnd3JytB2m9HXRP2HhHrE66
- 14ukuzLAVj88rAdM4U2288LrPbqM1wfdLA
- 3JfFwspVNFZ3t8Bfijf4ZGeec45fG3kdpS
- 3Fu7EkXXKL7phuZThKmbVzRbt5wh6V3Mrc
- 13Fu5ovsLwja3idPYUbFThdrtwN917EgQj

Chapter 4

Python Modules Developed *(Package Documentation)*

4.1 Module for Data Fetching

Code A.1 provides the module for fetching data from Bitcoin Core client. Bitcoin Core stores the data in .dat format and includes information about block height, hash, difficult, timestamp etc. Transactions are stored inside blocks and each transaction contains the script for verifying the transaction owner's secret key against his/her public key.
The official Bitcoin-RPC [Gar16] provides tool for interacting with the Bitcoin Core application and manipulate data. We forked the official python compatible Bitcoin-RPC client and modified it to gather data in our specified format. We stored data in blocks of 10 per JSON file. This was done in tune with the capabilities of our system.
Input:
start_Height: Starting height of the chain of blocks that the user wants to store in JSON format
end_Height: Ending height of the chain of blocks that the user wants to store in JSON format
step: Number of blocks to store per file
Output:
JSON files containing the blocks between the specified start and end height. Each file contains 10 blocks. It stores all the JSON files in a folder titled JSONFiles.

4.2 Module for making the Transaction (Tx) Network

Code A.2 provides the module for Making the network of all the Transactions happening over the Bitcoin Network. Each node represents a unique Tx. Its parent nodes are Tx from which it is claiming the money, while its children nodes are Tx to which money is being passed onto. The connections are identified using unique Transaction Ids (TxID) field in a Tx. The network is stored as a hash list of lists (adjacency list).
Input:
Point the code to the folder containing the JSON Files (by default, it is JSONFiles) and set the start height, end height and step variables accordingly.
Output:
Network of Transactions over the Bitcoin network between the specified heights.

4.3 Module for making the Public Key Network

Code A.3 provides the module for Making the network of all the Public Keys appearing over the Bitcoin Network. Each node represents a unique Public Key. Its parent nodes are Public Keys from which it has ever derived money, while its children nodes are Public Keys to which money has ever been passed onto.
An edge is added between two Public Keys if they appear in the input and output sections of the same Tx.
Input:
Point the code to the folder containing the JSON Files (by default, it is JSONFiles) and set the start height, end height and step variables accordingly.
Output:
Network of Public Keys over the Bitcoin network between the specified heights.

4.4 Module to create the Public-Key-Money Network

Code A.4 provides the module for making a weighted network of all the Public Keys appearing over the Bitcoin Network. Each node represents a

unique Public Key. Its parent nodes are Public Keys from which it has ever derived money, while its children nodes are Public Keys to which money has ever been passed onto.
An edge is added between two Public Keys if they appear in the input and output sections of the same Tx. The edges of this network are weighted. For a description of the weights used please refer to section 1.1.1
Input:
Point the code to the folder containing the JSON Files (by default, it is JSONFiles) and set the start height, end height and step variables accordingly.
Output:
Weighted network of Public Keys over the Bitcoin network between the specified heights.

4.5 Module to apply Multi-Input Heuristic on the PK Network

Applies the Multi-Input Heuristic as described in [Nic15][RH13][RS13].
Input:
Point the code to the folder containing the JSON Files (by default, it is JSONFiles) and set the start height, end height and step variables accordingly.
Output:
Network of ”Users” over the Bitcoin network between the specified heights.

4.6 Module to find Motifs in the PK network

Code A.6 provides the module for finding motifs in the PK Network. The motifs in the PK network depend heavily on the Tx Network. A n-input-m-output transaction creates the complete bipartite graph $K_{n,m}$ whose all edges are directed from the input PK to the output PK.
Input:
Point the code to the folder containing the JSON Files (by default, it is JSONFiles) and set the start height, end height and step variables accordingly.
Output:
Histogram of type of transactions versus their frequency.

4.7 Modules for converting the Transaction/Public Key Network to a SNAP Graph or Network

Codes A.7 and A.8 convert the given adjacency list to a SNAP Graph and Network types respectively.
SNAP Graph: Models an undirected graph with only one self loop per node and only one edge between any two given nodes.
SNAP Network: Models a directed multigraph with only one self loop per node but any number of directed edges between a pair of nodes.
Input:
name: Name of a PK/Tx Network
Output:
name.graph: SNAP Graph/Network realization of the PK/Tx Network.

4.8 Module to plot the degree distribution of the PK/Tx Network

Code A.9 uses SNAP subroutines to plot the degree distribution of the desired SNAP Network.
Input:
name: Name of a SNAP Network
outname: Primary name of the degree calculation files
Output:
The module outputs 6 files.
inDeg.outname.plt, inDeg.outname.plt: Prints the commands used to draw the plot.
inDeg.outname.png, inDeg.outname.png: Plots of the degree distribution, both in and out.
inDeg.outname.tab, inDeg.outname.tab: Degree distribution histogram.

4.9 Module to get the Clustering Coefficients of the desired Graph or Network

Codes A.10 and A.11 use SNAP subroutines to calculate the Clustering coefficients of the SNAP Graph/Network as defined in [WS98].
It calculates the global clustering coefficient and also plots the local clustering coefficients of vertices against their degree. Before calculating the Clustering Coefficients, we extract the maximum maximal connected component of the graph.

Input:
name: Name of a SNAP Network
Output:
Each module outputs 7 files.
ccf.X.plt, ccf.X.plt: Prints the commands used to draw the plot.
ccf.X.png, ccf.X.png: Plots of the local Clustering Coefficient. Also displays the global clustering coefficient.
inDeg.X.tab, outDeg.X.tab: Local Clustering Coefficient distribution histogram.
name.MaximalConnectedComponent.Graph: Graph of the Maximum Maximal Connected Component.
(where X=name_U for SNAP Graph and X=name_D for SNAP Network.)

4.10 Module to convert graph type

Code A.12 converts the Public-Key-Money network *(which is initially stored as a dictionary of lists)* to a sparse network in coordinate format. The non-zero entries' co-ordinate and value in the matrix is stored. We need this format to be able to use python modules numpy and scipy. This module also stores the mapping Public-Key → Coordinates. We need it to revert back to the names once the computation is finished.
Input:
name: Name of the Public-Key-Money network
Output:
Sparse_name.txt: Public-Key-Money Network in sparse co-ordinate format
indexing_Pagerank_name

4.11 Module to compute BitRank and Pagerank

Code A.13 computes BitRank *(see 3.2)* and PageRank *(see 3.2 or [Gle15] for a more in-depth treatment)* over the sparse weighted Public-Key-Money network created via code A.12.
Input:
name: Name of the Public-Key-Money network created using code A.12
Output:
simple_pagerank_scores_name.txt Contains the Pagerank scores of each node
weighted_pagerank_scores_name.txt Contains the BitRank score of each node.

4.12 Module to compare results

Code A.14 takes the results of the two ranking algorithms and sorts them in descending order of importance for convenient comparison.
Input:
name: Name of the score files created using code A.14
Output:
compare_rank_name.txt: Contains the Public Keys sorted in their decreasing order of importance.

Appendix A

Python Modules Developed *(Source Codes)*

A.1 Module for Data Fetching

Code A.1: To extract blocks from Bitcoin Core

```
def get_block_with_hash(hsh, api_code=None): #hsh is
   the hash of the block
    resource = 'block/{0}.json'.format(hsh) #querying
        bitcoinrpc to get the block
    if api_code is not None:
        resource += '&api_code=' + api_code
    response = net_util.call_api(resource)
    json_response = json.loads(response) #loading
       response in a JSON container
    return json_response #returning the caught
       response


if __name__ == '__main__':
    rpc_user = "<---Enter the rpc username here--->"
    rpc_password = "<---Enter the rpc password here--->
       "
    # rpc_user and rpc_password are set in the
       bitcoin.conf file
    rpc_connection = AuthServiceProxy("http://%s:%
       s@127.0.0.1:8332"%(rpc_user, rpc_password))

    script_dir = os.path.dirname(__file__)
    blocks = []
    start_height = int(raw_input("Enter␣start␣height:
        ␣"))
    end_height = int(raw_input("Enter␣end␣height:␣"))
    step = int(raw_input("Enter␣step␣size:␣"))
#step=10 #this is the default value

    #integer division is being performed
    if((end_height % step) != 0):
        end_height = (((end_height / step) + 1) *
            step)

    init = start_height
    for i in range(start_height+step, (end_height +
        1), step):#iterating over the number of blocks
        queried
        commands = [["getblockhash", height] for
            height in range(init, i)]
        block_hashes = rpc_connection.batch_(commands
            )
        #see for http rest batch processing support

        for hsh in block_hashes:
            #print hsh.
            block = get_block_with_hash(hsh)
            blocks.append(block)

        f=open('JSONFiles/data%i.json' %i,'w')
        json.dump(blocks,f,indent=4)
        f.close()
        blocks=[]
        print i
        init = i
```

A.2 Module for making the Transaction Network

Code A.2: To create the Transaction (Tx) network from the given blocks

```
import json

block1=[]
```

```
start=200000#start height
end=205000#end height
step=10#number of blocks stored per file
numb=end-start-10

for i in range(start+step,end,10):
 with open( 'JSONFiles/%i.json' %i,'r') as f:
  print 'Loading␣Block', i
  B=json.load(f)
  for j in range(0,1000):
   block1.append(B[j])

print 'Making␣Transaction␣Network'

count=0
KEcount=0
dTxNet={}
for k in range (0,numb):
 for i in range(0,len(block1[k]['tx'])):
  for j in range (0,len(block1[k]['tx'][i]['vin'])):
   try:
    if block1[k]['tx'][i]['vin'][j]['txid'] not in
       dTxNet:#capturing the input TxID
     dTxNet[block1[k]['tx'][i]['vin'][j]['txid']]=[]#
        Assigning the input TxID a new node
    dTxNet[block1[k]['tx'][i]['vin'][j]['txid']].
       append(block1[k]['tx'][i]['txid'])#making
       connection from previous TxID to this TxID
   except KeyError:
    KEcount=KEcount+1#counting bad transactions


print 'key', KEcount

f=open('tx%i.txt' %numb,'w')
json.dump(dTxNet,f,indent=4)#output the network
f.close()

print 'Output␣saved'
```

A.3 Module for making the Public Key Network

Code A.3: To create the Public Key network from the given blocks

```
import json

block1=[]

start=#start height
end=205000#end height
step=10#number of blocks stored per file
numb=end-start-10

for i in range(start+step,end,10):
#     filepath=path.relpath(')
 with open( 'JSONFiles/data%i.json' %i,'r') as f:
  print 'Loading␣Block', i
  B=json.load(f)
  for j in range(0,10):
   block1.append(B[j])




dOutTxid={}#dictionary to store Tx->Output PK
for k in range (0,numb):
 for i in range(0,len(block1[k]['tx'])):
  for j in range (0,len(block1[k]['tx'][i]['vout'])):
   try:
    if block1[k]['tx'][i]['txid'] not in dOutTxid:
     dOutTxid[block1[k]['tx'][i]['txid']]=[]
    dOutTxid[block1[k]['tx'][i]['txid']].append(
       block1[k]['tx'][i]['vout'][j]['scriptPubKey']['
       addresses'][0])
    if len(block1[k]['tx'][i]['vout'][j]['
       scriptPubKey']['addresses'])>1:
     print 'Possible␣Escrow' + block1[k]['tx'][i]['
        txid']
   except KeyError:
    continue
```

```
dInpTxid={}#dictionary to store Tx->Input PK
count=0;
for k in range (0,numb):
 dInpTxid[block1[k]['tx'][0]['txid']]=[]
 dInpTxid[block1[k]['tx'][0]['txid']].append('
    coinbase')#dummy input for miner
 for i in range(0,len(block1[k]['tx'])):
  for j in range (0,len(block1[k]['tx'][i]['vin'])):
   try:
    if block1[k]['tx'][i]['txid'] not in dInpTxid:
     dInpTxid[block1[k]['tx'][i]['txid']]=[]
    txid=block1[k]['tx'][i]['vin'][j]['txid']#the
       transaction id of the transaction this input is
        taken from
    vout=block1[k]['tx'][i]['vin'][j]['vout']#the
       index of the output used for this input
    dInpTxid[block1[k]['tx'][i]['txid']].append(
       dOutTxid[txid][vout])
   except KeyError:
    continue


dPkNet={}
for txid in dInpTxid:#iterate over the Transactions
 try:
  pks_in_txid=dOutTxid[txid]
  for in_addr in dInpTxid[txid]:
   for out_pk in pks_in_txid:
    if in_addr not in dPkNet:#assign a node to Public
        Key if not previously encountered.
     dPkNet[in_addr]=[]
    dPkNet[in_addr].append(out_pk)#make output Public
        Keys of this Tx, the out neighbours of current
        PK.
 except KeyError:
  pass



f=open('pk_%i_%i.txt' %(start,numb),'w')
json.dump(dPkNet,f,indent=4)
```

```
f.close()

print 'Done'
```

A.4 Module for making the Public-Key-Money Network

Code A.4: To create the Public-Key-Money network

```
import json

block1=[]


def pkWN(start,end,step,numb,name):
 for i in range(start+step,end+step,step):
#     filepath=path.relpath(')
  with open( 'JSONFiles/data%i.json' %i,'r') as f:
   print 'Loading␣Block', i
   B=json.load(f)
   for j in range(0,10):
    block1.append(B[j])

 dOutTxid={}
 for k in range (0,numb):
  for i in range(0,len(block1[k]['tx'])):
   for j in range (0,len(block1[k]['tx'][i]['vout']))
      :
    try:
     if block1[k]['tx'][i]['txid'] not in dOutTxid:
      dOutTxid[block1[k]['tx'][i]['txid']]=[]
     dOutTxid[block1[k]['tx'][i]['txid']].append([
        block1[k]['tx'][i]['vout'][j]['scriptPubKey'][
        'addresses'][0],block1[k]['tx'][i]['vout'][j][
        'value']])
     if len(block1[k]['tx'][i]['vout'][j]['
        scriptPubKey']['addresses'])>1:
      print 'Possible␣Escrow' + block1[k]['tx'][i]['
         txid']
    except KeyError:
     continue
```

```
'''
f=open('testDOUT.txt','w')
json.dump(dOutTxid,f,indent=4)
f.close()
'''

dInpTxid={}
count=0;
for k in range (0,numb):
 dInpTxid[block1[k]['tx'][0]['txid']]=[]
 dInpTxid[block1[k]['tx'][0]['txid']].append(['
    coinbase',25])#dummy input for miner
 for i in range(0,len(block1[k]['tx'])):
  for j in range (0,len(block1[k]['tx'][i]['vin'])):
   try:
    if block1[k]['tx'][i]['txid'] not in dInpTxid:
     dInpTxid[block1[k]['tx'][i]['txid']]=[]
    txid=block1[k]['tx'][i]['vin'][j]['txid']#the
       transaction id of the transaction this input
       is taken from
    vout=block1[k]['tx'][i]['vin'][j]['vout']#the
       index of the output used for this input
    dInpTxid[block1[k]['tx'][i]['txid']].append([
       dOutTxid[txid][vout][0],dOutTxid[txid][vout
       ][1]])
   except KeyError:
    continue
   except IndexError:
    continue


'''
f=open('testDIN','w')
json.dump(dInpTxid,f,indent=4)
f.close()
'''
dPkNet={}
for txid in dInpTxid:
 try:
  #Total_data_out_txid=dOutTxid[txid]
  s=0

  for data_out_txid in dOutTxid[txid]:
   s=s+data_out_txid[1]    #base weight for
      distribution of edge weight

  for data_in_txid in dInpTxid[txid]:
   in_addr = data_in_txid[0]
   in_value= data_in_txid[1]
   if in_addr not in dPkNet:
    dPkNet[in_addr]=[]
   for data_out_txid in dOutTxid[txid]:
    out_addr=  data_out_txid[0]
    out_value= data_out_txid[1]
    edge_weight= in_value*out_value/s
    dPkNet[in_addr].append([out_addr,edge_weight])
 except KeyError:
  continue



f=open(name,'w')
json.dump(dPkNet,f,indent=4)
f.close()

print 'Done'
```

A.5 Module to apply Multi-Input Heuristic on the PK Network

Code A.5: To apply Multi-Input Heuristic on the PK Network

```
import json

block1=[]

start=200000
end=205000
step=10
numb=end-start-10

for i in range(start+step,end,10):
 with open( 'JSONFiles/data%i.json' %i,'r') as f:
  print 'Loading Block', i
  B=json.load(f)
  for j in range(0,10):
   block1.append(B[j])

'''User Naming Convention: A pk which is encountered
   for the first time is allocated to the username
"Hash of the transaction" in which it was found first
   , subsequent occurences and the associated
public keys are mapped to that same user'''


dOutTxid={}
for k in range (0,numb):
 for i in range(0,len(block1[k]['tx'])):
  for j in range (0,len(block1[k]['tx'][i]['vout'])):
   try:
    if block1[k]['tx'][i]['txid'] not in dOutTxid:
     dOutTxid[block1[k]['tx'][i]['txid']]=[]
    dOutTxid[block1[k]['tx'][i]['txid']].append(
       block1[k]['tx'][i]['vout'][j]['scriptPubKey']['
       addresses'][0])
    if len(block1[k]['tx'][i]['vout'][j]['
       scriptPubKey']['addresses'])>1:
     print 'Possible␣Escrow␣' + block1[k]['tx'][i]['
        txid']
   except KeyError:
    continue




dInpTxid={}
count=0;
for k in range (0,numb):
 dInpTxid[block1[k]['tx'][0]['txid']]=[]
 dInpTxid[block1[k]['tx'][0]['txid']].append('
    coinbase')#dummy input for miner
 for i in range(0,len(block1[k]['tx'])):
  for j in range (0,len(block1[k]['tx'][i]['vin'])):
   try:
    if block1[k]['tx'][i]['txid'] not in dInpTxid:
     dInpTxid[block1[k]['tx'][i]['txid']]=[]
    txid=block1[k]['tx'][i]['vin'][j]['txid']#the
       transaction id of the transaction this input is
```

```
        taken from
    vout=block1[k]['tx'][i]['vin'][j]['vout']#the
       index of the output used for this input
    dInpTxid[block1[k]['tx'][i]['txid']].append(
       dOutTxid[txid][vout])
   except KeyError:
    continue

dPkUser={}
#start of the multi Input heuristic
for txid in dInpTxid:
 match=0
 for pk in dInpTxid[txid]:
  if pk in dPkUser:
   match=1
   user=dPkUser[pk]
 if match==1:
  for pk in dInpTxid[txid]:
   dPkUser[pk]=user
 else:
  for pk in dInpTxid[txid]:
   dPkUser[pk]=txid
#end of the multi Input heuristic



dPkNet={}
for txid in dInpTxid:
 try:
  pks_in_txid=dOutTxid[txid]
  for in_addr in dInpTxid[txid]:
   if in_addr in dPkUser:
    in_addr=dPkUser[in_addr]#converting input pk to
       user
   for out_pk in pks_in_txid:
    if out_pk in dPkUser:
     out_pk=dPkUser[out_pk]#converting output pk to
        user
    if in_addr not in dPkNet:
     dPkNet[in_addr]=[]
    dPkNet[in_addr].append(out_pk)
 except KeyError:
  pass
```

```
f=open('upMIH%i.txt' %numb,'w')
json.dump(dPkNet,f,indent=4)
f.close()

print 'Done'
```

A.6 Module to find Motifs in the PK network

Code A.6: To find various Motifs in the PK network

```
import json

block1=[]

start=200000#start height
end=205000#end height
step=10#number of blocks stored per file
numb=end-start-10

for i in range(start+step,end,10):
 with open('JSONFiles/data%i.json' %i,'r') as f:
  print 'Loading␣Block', i
  B=json.load(f)
  for j in range(0,10):
   block1.append(B[j])

print 'Calculating␣Motifs'

dMotif={}#dictionary to store motifs
count=0
for k in range (0,numb):
 for i in range(0,len(block1[k]['tx'])):
  count=count+1
  z=str(len(block1[k]['tx'][i]['vin']))+':'+str(len(
     block1[k]['tx'][i]['vout']))
  if z not in dMotif:
   dMotif[z]=1
  else:
   dMotif[z]=dMotif[z]+1
```

```
for entry in dMotif:
 if dMotif[entry]>1000:
  print entry+' '+str(dMotif[entry])
print count

f=open('Motif%i.txt' %numb,'w')
json.dump(dMotif,f,indent=4,sort_keys=True)#output
   the network
f.close()

print 'Output saved'
```

A.7 Module for converting the Transaction/Public Key Network to a SNAP Graph

Code A.7: To convert Transaction/Public Key Network to SNAP Graph format

```
from snap import *
import json

name='upMIH4990'

with open('%s.txt' %name ,'r') as f:
 d1=json.load(f)

d2={}#dictionary node->count
d3={}#dictionary count->node
G=TUNGraph.New()

count=0;
for entry in d1:
 d2[entry]=count
 G.AddNode(d2[entry])
 count=count+1
 for something in d1[entry]:
  if something not in d2:
   d2[something]=count
   G.AddNode(d2[something])
   count=count+1
```

```
print count

for entry in d1:
 for something in d1[entry]:
  G.AddEdge(d2[entry],d2[something])

print 'Made the graph'
F = TFOut('%s.graph' %name)
G.Save(F)
F.Flush()
```

A.8 Module for converting the Transaction/Public Key Network to a SNAP Network

Code A.8: To convert Transaction/Public Key Network to SNAP Network format

```
from snap import *
import json

name='upMIH4990'

with open('%s.txt' %name ,'r') as f:
 d1=json.load(f)

d2={}#dictionary node->count
d3={}#dictionary count->node
G=TNEANet.New()

count=0;
for entry in d1:
 d2[entry]=count
 G.AddNode(d2[entry])
 count=count+1
 for something in d1[entry]:
  if something not in d2:
   d2[something]=count
   G.AddNode(d2[something])
   count=count+1

print count
```

```
for entry in d1:
 for something in d1[entry]:
  G.AddEdge(d2[entry],d2[something])

print 'Made the graph'
F = TFOut('%s.graph' %name)
G.Save(F)
F.Flush()
```

A.9 Module to plot the degree distribution of the desired Network

Code A.9: To plot the degree distribution of the Transaction/Public Key Network

```
from snap import *
import json

name="<--Enter the SNAP network name here-->"

F = TFIn('%s.graph' %name)
G = TNEANet.Load(F)
print 'Loaded Graph'

outname="<--Enter the output file name-->"
PlotInDegDistr(G, outname,"Bitcoin In Degree")#plot
   in-degree distribution
PlotOutDegDistr(G, outname,"Bitcoin Out Degree")#plot
    out-degree distribution

print 'done'
```

A.10 Module to get the Clustering Coefficients of the desired Graph

Code A.10: To get the Global Clustering Coefficients and plot the Local Clustering Coefficients of the desired Graph

```
from snap import *
import json
```

```

NIdV = TIntV()

name='<--Enter input file name here-->'

F = TFIn('%s.graph' %name)
G = TUNGraph.Load(F)
print 'Loaded Graph'

Components = TCnComV()
GetSccs(G, Components)

for i in Components[0]:
 NIdV.Add(i)#get vertices in this component

print 'Got Components'

SG=GetSubGraph(G,NIdV)

print 'Made subgraph using the maximum maximal 
   connected component'

PlotClustCf(SG, "%s_U" %name, "Bitcoin")#plot
   clustering coefficients

print 'Plotted the coefficients'

F = TFOut('%sMaximalConnectedComponent.graph' %name)#
   save the maximum maximal connected component
SG.Save(F)
F.Flush()

#print SG.len()
print 'Done'
```

A.11 Module to get the Clustering Coefficients of the desired Network

Code A.11: To get the Global Clustering Coefficients and plot the Local Clustering Coefficients of the desired Network

```
from snap import *
```

```
import json

NIdV = TIntV()

name='<--Enter input file name here-->'

F = TFIn('%s.graph' %name)
G = TNEANet.Load(F)
print 'Loaded Graph'

Components = TCnComV()
GetSccs(G, Components)#Extract components

print 'Loaded Graph'

for i in Components[0]:
 NIdV.Add(i)#get vertices in this component

SG=GetSubGraph(G,NIdV)

print 'Got the maximum maximal connected component'

PlotClustCf(SG, "%s_D" %name, "Bitcoin")
F = TFOut('%sMaximalConnectedComponent.graph' %name)#
    save the maximum maximal connected component
SG.Save(F)
F.Flush()

#print SG.len()
''' '''
```

A.12 Module to convert Public-Key-Money network's sparse format

Code A.12: To convert Public-Key-Money network's sparse format from dictionary-of-lists to co-ordinate-value triples. *(compatible with numpy and scipy)*

```
import json
import csv

def NWS(name):
```

```
with open(name,'r') as f:
 d1=json.load(f)

d2={}#dictionary node->count
d3={}#dictionary count->node

count=0
for entry in d1:
 if entry not in d2:
  count=count+1
  d2[entry]=count
  d3[count]=entry
  #G.AddNode(d2[entry])
 for something in d1[entry]:
  if something[0] not in d2:
   count=count+1
   d2[something[0]]=count
   d3[count]=something[0]
   #G.AddNode(d2[something[0]])


print 'nodes␣' + str(count)
NumberOfNodes=count

f=open('Sparse_%s' %name, 'w')
sw = csv.writer(f, delimiter='␣')

sw.writerow([NumberOfNodes])#printing the shape of
   our sparse matrix


d4={}#pair->unique id->money (to keep a list of the
   pairs we have used)
for entry in d1:
 for something in d1[entry]:
  uniqueId=d2[entry]*count+d2[something[0]]#see how
     this becomes the unique id for this 2-d array
  money=0
  if uniqueId not in d4:
   d4[uniqueId]=something[1]
  else:
   d4[uniqueId]=something[1]+d4[uniqueId]
```

```
d5={}#to keep a list of pairs we have outputted
for entry in d1:
 for something in d1[entry]:
  uniqueId=d2[entry]*count+d2[something[0]]
  if uniqueId not in d5:
   sw.writerow([d2[entry],d2[something[0]],d4[
      uniqueId]])
   d5[uniqueId]=""
f.close()

f=open('indexing_Pagerank_%s' %name,'w')
json.dump(d3,f,indent=4)
f.close()
```

A.13 Module to get BitRank and Pagerank of Public Keys

Code A.13: To compute BitRank and Pagerank scores of Public Keys over the Bitcoin Network

```
import json
import csv
import numpy as np
from scipy.sparse import csr_matrix
from scipy.sparse import csc_matrix

def PRW(name):
 f=open('Sparse_%s' %name, 'r')
 sw = csv.reader(f, delimiter='␣')

 with open('indexing_Pagerank_%s' %name,'r') as f1:
  d3=json.load(f1)
 maxIter=20
 s=0.8
 maxerr=0.01
 r=[]
 c=[]
 d=[]
 sd=[]
 flag=0
 for line in sw:
   if flag==0:
    n=int(line[0])
    flag=1
    continue
   else:
    r.append(int(line[0])-1)#subtracting 1 in
        convention with numpy format whose index starts
        with 0.
    c.append(int(line[1])-1)#subtracting 1 in
        convention with numpy format whose index starts
        with 0.
    d.append(float(line[2]))#Weighted Pagerank
    sd.append(1.0)#simple Pagerank
 #print r,c,d
 f.close()

 row = np.array(r)
 col = np.array(c)
 data = np.array(d)
 simpleData=np.array(sd)
 G=csr_matrix((data, (row, col)), shape=(n, n))#
    making a sparse scipy matrix
 SG=csr_matrix((simpleData, (row, col)), shape=(n, n)
    )#making a sparse scipy matrix

 #print SG.toarray()

#********************-------Weighted--Pagerank--
   computation------******************

 rsums = np.array(G.sum(1))[:,0]#computing row sum
 for item in range(0,G.nnz):
  data[item]=data[item]/rsums[row[item]]#normalizing
     to make markov matrix

 G=csc_matrix((data, (row, col)), shape=(n, n))
 #print G[946,:]

 sink=(rsums==0)#recognizing sinks

 # Compute pagerank r until we converge
 ro, r = np.zeros(n), np.ones(n)/n
 ite=0
 while np.sum(np.abs(r-ro)) > maxerr and ite <
    maxIter:
  ro = r.copy()
 # calculate each pagerank at a time
  for i in xrange(0,n):
   # inlinks of state i
   Ii = np.array(G[:,i].todense())[:,0]
   # account for sink states
   Si = sink / float(n)
   # account for teleportation to state i
   Ti = np.ones(n) / float(n)
   r[i] = ro.dot( Ii*s + Si*s + Ti*(1-s) )
  ite+=1
  print ite
  # return normalized pagerank

 d1={}
 #normFactor=sum(r)

 for i in range(0,len(r)):
  d1[d3[str(i+1)]]=r[i]#Re-Indexing for user ease

 with open('weighted_pagerank_scores_%s' %name,'w')
    as f:
  json.dump(d1,f,indent=4)
#********************-------Weighted---Pagerank---
   computation---Over------******************

 print 'Computed weighted pagerank'

#********************-------Simple---Pagerank---
   computation------******************

 rsums = np.array(SG.sum(1))[:,0]#computing row sum
 for item in range(0,SG.nnz):
  simpleData[item]=simpleData[item]/rsums[row[item]]#
     normalizing to make markov matrix

 SG=csc_matrix((simpleData, (row, col)), shape=(n, n)
    )
 sink=(rsums==0)#recognizing sinks

 # Compute pagerank r until we converge
```

```
ro, r = np.zeros(n), np.ones(n)/n
ite=0
while np.sum(np.abs(r-ro)) > maxerr and ite <
   maxIter:
 ro = r.copy()
# calculate each pagerank at a time
 for i in xrange(0,n):
  # inlinks of state i
  Ii = np.array(SG[:,i].todense())[:,0]
  # account for sink states
  Si = sink / float(n)
  # account for teleportation to state i
  Ti = np.ones(n) / float(n)
  r[i] = ro.dot( Ii*s + Si*s + Ti*(1-s) )
 ite+=1
 print ite
 # return normalized pagerank

d2={}
normFactor=sum(r)

for i in range(0,len(r)):
 d2[d3[str(i+1)]]=r[i]#Re-Indexing for user ease

with open('simple_pagerank_scores_%s' %name,'w') as
   f:
 json.dump(d2,f,indent=4)

#********************-------Simple--Pagerank--
   computation--Over-------******************

print 'Computed simple pagerank'
```

A.14 Module to compare the ranks of Public Keys in PageRank and BitRank

Code A.14: To compare the ranks of Public Keys in PageRank and BitRank

```
import json
def BSR(name):
 f1=open('simple_pagerank_scores_%s' %name,'r')
 f2=open('weighted_pagerank_scores_%s' %name,'r')
```

```
l1=[]
l2=[]

d1=json.load(f1)
d2=json.load(f2)

for key in d1:
 l1.append(key)

print len(l1)

for j in range(0,len(l1)):
 for i in range(0,len(l1)-1-j):
  if d1[l1[i]]<d1[l1[i+1]]:
   temp=l1[i+1]
   l1[i+1]=l1[i]
   l1[i]=temp

print l1[0]

for key in d2:
 l2.append(key)

print len(l2)

for j in range(0,len(l2)):
 for i in range(0,len(l2)-1-j):
  if d2[l2[i]]<d2[l2[i+1]]:
   temp=l2[i+1]
   l2[i+1]=l2[i]
   l2[i]=temp

print l2[0]

f1.close()
f2.close()

f3=open('compare_rank_%s' %name, 'w')

for i in range(0,len(l1)):
 f3.write(l1[i]+' '+l2[i]+' '+str(d1[l1[i]])+' '+str
    (d2[l2[i]])+'\n')
```

References

[10] *Bitcoin Wiki.* April 14,2010. URL: `https://en.bitcoin.it/wiki/Main_Page`.

[Sim55] Herbert A Simon. "On a class of skew distribution functions". In: *Biometrika* 42.3/4 (1955), pp. 425–440.

[WS98] Duncan J Watts and Steven H Strogatz. "Collective dynamics of 'small-world'networks". In: *nature* 393.6684 (1998), pp. 440–442.

[Pag+99] Lawrence Page et al. *The PageRank citation ranking: Bringing order to the web.* Tech. rep. Stanford InfoLab, 1999.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. "The elliptic curve digital signature algorithm (ECDSA)". In: *International Journal of Information Security* 1.1 (2001), pp. 36–63.

[Nak08] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system.* 2008.

[And09] K Andreas. *Valuation of Network Effects in Software Markets: A Complex Networks Approach.* 2009.

[BP12] Sergey Brin and Lawrence Page. "Reprint of: The anatomy of a large-scale hypertextual web search engine". In: *Computer networks* 56.18 (2012), pp. 3825–3833.

[MSK12] Ali Masoudi-Nejad, Falk Schreiber, and ZRM Kashani. "Building blocks of biological networks: a review on major network motif discovery algorithms". In: *IET systems biology* 6.5 (2012), pp. 164–174.

[RH13] Fergal Reid and Martin Harrigan. "An analysis of anonymity in the bitcoin system". In: *Security and privacy in social networks.* Springer, 2013, pp. 197–223.

[RS13] Dorit Ron and Adi Shamir. "Quantitative analysis of the full bitcoin transaction graph". In: *International Conference on Financial Cryptography and Data Security.* Springer. 2013, pp. 6–24.

[Fra14] Pedro Franco. *Understanding Bitcoin: Cryptography, engineering and economics*. John Wiley & Sons, 2014.

[FKP15] Michael Fleder, Michael S Kester, and Sudeep Pillai. "Bitcoin transaction graph analysis". In: *arXiv preprint arXiv:1502.01657* (2015).

[Gle15] David F Gleich. "PageRank beyond the Web". In: *SIAM Review* 57.3 (2015), pp. 321–363.

[Mar15] Sergio Ivan Marcin. "Bitcoin Live: Scalable system for detecting bitcoin network behaviors in real time." In: (2015).

[Nic15] Jonas David Nick. "Data-Driven De-Anonymization in Bitcoin". PhD thesis. ETH-Zürich, 2015.

[16] *Blockchain.info*. 2016. URL: `https://blockchain.info`.

[Gar16] Jeff Garzik. *python-bitcoinrpc 1.0*. 2016. URL: `http://www.github.com/jgarzik/python-bitcoinrpc`.

[HF16] Martin Harrigan and Christoph Fretter. "The Unreasonable Effectiveness of Address Clustering". In: *arXiv preprint arXiv:1605.06369* (2016).

[LS16] Jure Leskovec and Rok Sosič. "SNAP: A General-Purpose Network Analysis and Graph-Mining Library". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 8.1 (2016), p. 1.

[Wik16a] Wikipedia. *Clustering coefficient — Wikipedia, The Free Encyclopedia*. [Online; accessed 29-October-2016]. 2016. URL: `https://en.wikipedia.org/w/index.php?title=Clustering_coefficient&oldid=746851508`.

[Wik16b] Wikipedia. *Degree distribution — Wikipedia, The Free Encyclopedia*. [Online; accessed 6-October-2016]. 2016. URL: `%5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Degree_distribution&oldid=742831289%7D`.

[Wik16c] Wikipedia. *Network motif — Wikipedia, The Free Encyclopedia*. [Online; accessed 21-July-2016]. 2016. URL: `https://en.wikipedia.org/w/index.php?title=Network_motif&oldid=730873013`.