Bitcoin Transactions Network Analysis : Actors Identification and Predictions

References

- Course : Bitcoin Network analysis, 2020, R. Cazabet, Université Paris 1 Panthéon-Sorbonne, Université de Lyon
- Brin, S. and Page, L. (1998) The Anatomy of a Large-Scale Hypertextual Web Search Engine. In: Seventh International World-Wide Web Conference (WWW 1998), April 14-18, 1998, Brisbane, Australia.
- Kondor et al., Do the Rich Get Richer? An Empirical Analysis of the Bitcoin Transaction Network. PLOS ONE 9(5): e97205.
- Möser, Böhmen and Breuker, Inquiry into money laundering tools in the Bitcoin ecosystem, Conference: 2013 eCrime Researchers Summit (eCRS)
- Dorit Ron and Adi Shamir, Quantitative Analysis of the Full Bitcoin Transaction Graph, 2012
- Zhou, T., Lü, L., & Zhang, Y. C. (2009). Predicting missing links via local information. The European Physical Journal B, 71(4), 623-630
- node2vec: Scalable Feature Learning for Networks, 2016, A. Grover and J. Leskovec, Stanford University

Import packages

In [4]:
```
import os
import pandas as pd
import numpy as np
import networkx as nx
from networkx import read_graphml
import matplotlib.pyplot as plt
import seaborn as sns
from operator import itemgetter
import random
import datetime
from matplotlib.pyplot import figure
from numpy import mean
from cdlib import algorithms, viz, NodeClustering, evaluation
```

Import data

http://cazabetremy.fr/Teaching/bitcoinClass/time_split/year=2014/month=2/day=7/ (http://cazabetremy.fr/Teaching/bitcoinClass/time_split/year=2014/month=2/day=7/)

The file imported corresponds to 7 February 2014. Format is "parquet".

- value: value in satoshi.
- time: timestamp of the transaction.
- src_identity: source of the transaction.
- dst_identity: destination of the transaction.
- PriceUSD: Value of a bitcoin in USD for that day.

The id of actors can be: a name (known actor), an integer (group of addresses, "wallet"), a bitcoin address (address belongs to no cluster).

7 February 2014 :

- Hacking day
- The victim of a massive hack, Mt. Gox lost about 740,000 bitcoins (6% of all bitcoin in existence at the time), valued at the equivalent of €460 million

Setting our directory

Muge directory

In [7]:
```
os.chdir(r'/Users/mugefirsat/Downloads')
data = pd.read_parquet("7fev2014.parquet",engine='pyarrow')
```

Chloe directory

In [2]:
```
os.chdir(r'/Users/Daudenthun/Documents/M2/Network analysis/Data')
data = pd.read_parquet("7fev2014.parquet",engine='pyarrow')
```

Louis directory

In [97]:
```
os.chdir(r'/Users/Louis/Downloads')
data = pd.read_parquet("7fev2014.parquet",engine='pyarrow')
```

Data preparation

In [98]:
```
data.shape
```

Out[98]:
```
(207116, 5)
```

The initial dataset is composed of 207 116 rows (transactions) and 5 columns (value, time, source, destination, priceUSD).

- value : value in satoshi
- time : timestamp of the transaction
- src_identity : source of the transaction
- dst_identity : destination of the transaction
- PriceUSD : Value of a bitcoin in USD for that day.

The id of actors can be: a name (known actor), an integer (group of addresses, "wallet"), a bitcoin address (address belongs to no cluster).

In [99]:
```
data.head()
```

Out[99]:

	value	time	src_identity	dst_identity	
0	10860	1391783589	73346613	nonstandard141df9d18f92c7efe8e5cd927c2e9be7c78...	711
1	10860	1391784042	ePay.info	nonstandarde7284c974affd2b089a7df844876067ba49...	711
2	10860	1391810470	22801559	nonstandard3ec04a56a71c58b3ee99a71ebfa9ffecb17...	711
3	10860	1391730636	69408508	nonstandardfb0bdcc6b8e9ecec58cd04b89cb3dfd54b3...	711
4	10860	1391783589	82818	nonstandardd676f0143edff9d402846eb4f568a9bd1f8...	711

Rename columns

In [100]:
```
data = data.rename(columns = {'src_identity':'Source', 'dst_identit
```

Add the date

In [101]:
```
data['Date'] = pd.to_datetime(data['Time'],unit='s')
```

Add the hour

In [102]:
```
data['Hour'] = data.apply(lambda x: x['Date'].hour, axis=1)
```

Convert to numeric

In [103]:
```
data['PriceUSD'] = pd.to_numeric(data['PriceUSD'])
```

Add the equivalent bitcoin amount and dollar amount

In [104]:
```
data['Bitcoin'] = pd.DataFrame(data['Montant']*0.00000001)
data['Dollar'] = data.apply(lambda x: x['Bitcoin']*x['PriceUSD'], a
```

For convinience, we reduce the length of addresses.

In [105]:
```
data['Destination'] = data['Destination'].apply(lambda x: x[:20])
```

Data informations

Description of the transactions amount in dollar

In [106]:
```
data['Dollar'].describe().apply(lambda x: format(x, 'f'))
```

Out[106]:
```
count      207116.000000
mean         2979.761162
std         53682.132124
min             0.000000
25%             7.752252
50%            35.875815
75%           285.169151
max      14299594.707946
Name: Dollar, dtype: object
```

The average amount sent by transaction is 2 979 dollars with a median of 35 dollars, the minimal amount sent is 0 dollars and the maximal amount sent is 14 299 594 dollars. Just to recall that at this date, a bitcoin was worth 711.57 dollars.

Sorted by amount

In [107]:
```
sorted_asc = data.sort_values("Dollar", ascending=[False])
sorted_asc[['Dollar','Source','Destination','Date','Hour']].head()
```

Out[107]:

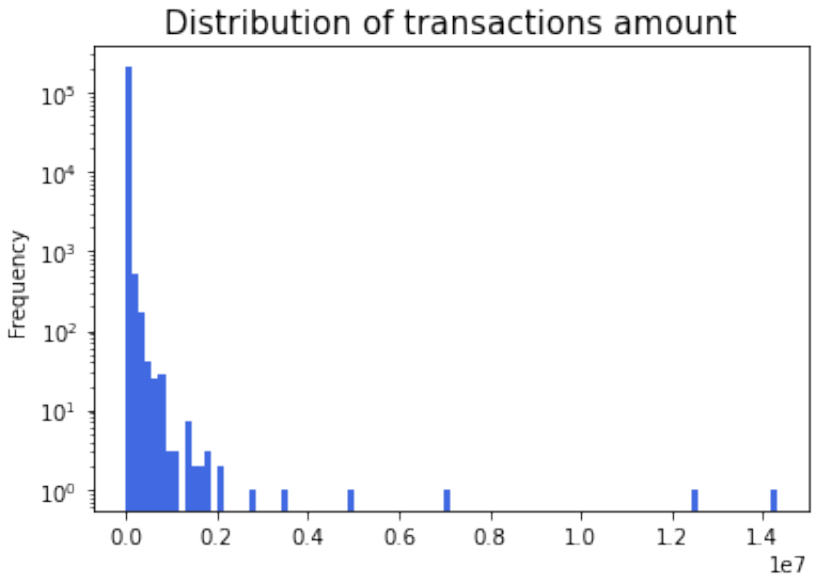
	Dollar	Source	Destination	
43878	1.429959e+07	72539960	72539960	2 (06:
43879	1.252060e+07	72539960	72539960	2 (07:
160505	7.115713e+06	72539960	Bitstamp.net-old	2 (06:
38366	4.891207e+06	Bitstamp.net-old	1GTPMZyBZidSwifYcdzH	2 (06:
60131	3.452625e+06	1GTPMZyBZidSwifYcdzHZhNzyMkJi7FZgy	1Gm8Ag9aVujX28Yc6mm5	2 (06:

Above, we can see the five most important transactions by amount. It's interesting to highlight that a transaction from Bitstamp.net-old to the public adsress "1GTPMZyBZidSwifYcdzHZhNzyMkJi7FZgy" and this same public adress send a large amount to another public address "1Gm8Ag9aVujX28Yc6mm5b8eNJKp5KjD6e".

Distribution of values of transaction

In [108]:
```
ax = data["Dollar"].plot.hist(stacked=True, bins=100, color = "roya
ax.set_yscale("log")
plt.title("Distribution of transactions amount", size=15)
```

Out[108]:
```
Text(0.5, 1.0, 'Distribution of transactions amount')
```

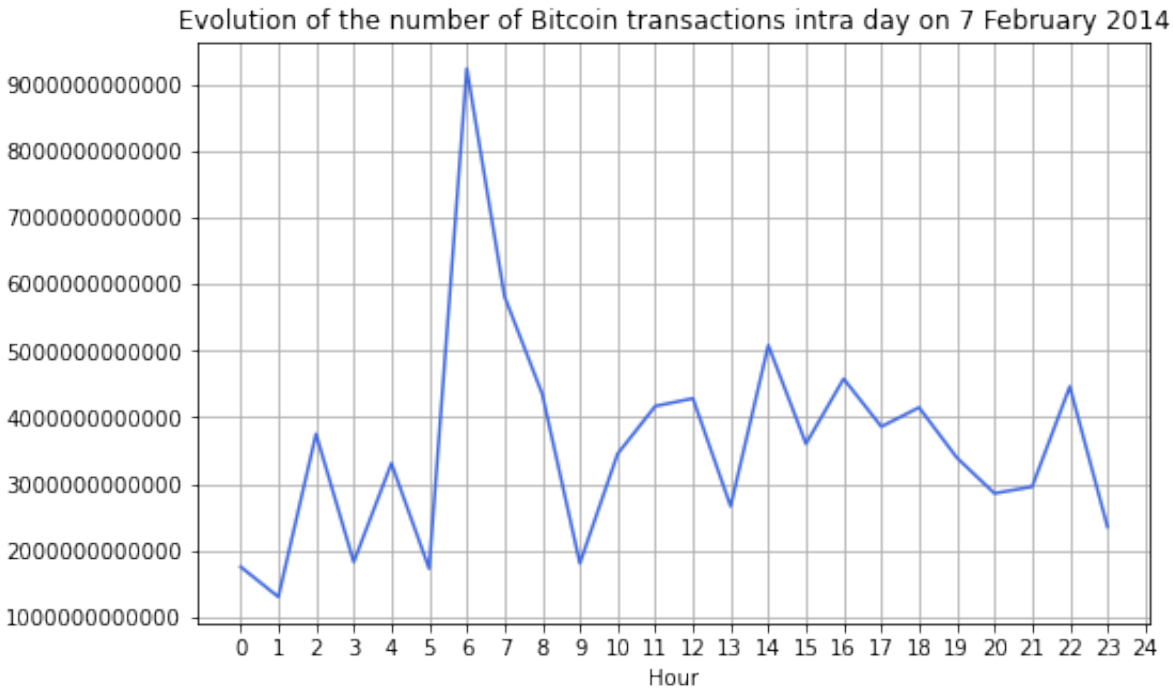


Here we can see the distribution of the dollar amount for the transactions of the 7th of febuary 2014. They are concentrated between 0 and 2 000 000 dollars.

Evolution of the number of transaction per hour

In [109]:
```
plt.figure(figsize = (8,5))
data.groupby(data["Hour"])["Montant"].sum().plot(color="royalblue")
plt.ticklabel_format(axis="y", style="plain")
plt.xticks(range(0,25))
plt.grid(True)
plt.title('Evolution of the number of Bitcoin transactions intra da
```

Out[109]:
```
Text(0.5, 1.0, 'Evolution of the number of Bitcoin transactions in
tra day on 7 February 2014')
```



In [110]:
```
print(data.groupby(data["Hour"])["Montant"].sum().describe().apply(
```

```
count    2.400000E+01
mean     3.613811E+12
std      1.675063E+12
min      1.299850E+12
25%      2.585641E+12
50%      3.525616E+12
75%      4.301000E+12
max      9.241505E+12
Name: Montant, dtype: object
```

The graphic below shows us the evolution of the number of Bitcoin intra day transactions on the 7 Feburary 2014.
Between 5am and 6am (between 05:00-06:59), the number of transactions skyrockets. At 6 am (between 06:00-06:59), a pick can be seen. Between 6 am and 9 am (between 06:00-09:59), the number of transactions increases and at 9am (between 09:00-09:59), one of the lowest transactions number is observed.
The avarage intra day transaction number is arround 3.61 x 10^12.
The lowest intra day transaction number that is 1.29 x 10^12 and is observed at between 01:00-01:59.

Overall network

Create an empty graph

In [195]:
```
G = nx.Graph()
```

Define edges and nodes

In [196]:
```
G = nx.from_pandas_edgelist(data, 'Source','Destination',["Dollar"]
```

Descriptive Informations

In [197]:
```
n = G.number_of_nodes()
m = G.number_of_edges()
```

In [201]:
```
print("Number of nodes :", str(n))
print("Number of edges :", str(m))
print("Number of connected components :",str(nx.number_connected_co
print('Density :', nx.density(G))
print ("avg degree: "+str(mean([n for n in dict(nx.degree(G)).value
print('Transitivity :', nx.transitivity(G))
```

```
Number of nodes : 94539
Number of edges : 139161
Number of connected components : 4329
Density : 3.114082557957132e-05
avg degree: 2.943991368641513
Transitivity : 0.0012238298379050325
```

On the 7th of febraury 2014, there were 94 539 nodes and 139 161 edges on the bitcoin network. Recall that in our bitcoin network representation, an edge represents a trasaction. Moreover, a Bitcoin holder can send his/her Bitcoin(s) to multiple receviers. In that case, one transaction can generate multiple edges. In addition, people can make reccurent transactions or also multiple receivers transactions. For instance, if Alice sends x Bitcoin to Bob at 6am and Bob sends Alice z Bitcoin at 10pm, thiw will generate 2 transactions but only 1 edge, this is a recurrent transaction. If Bob sends x Bitcoin to Alice, Chloé, Louis and Müge at once, this will generate 5 transactions (1 change and 4 sending) but only 4 edges. Thus, if one trasaction had had only one sender and one recevier and a couple of sender/receiver would have realised only one transaction which each other, this would have lead to 94 539 (nodes) * 2 = 189 078 edges. The fact that the number of edges are less than 189 078 underlines that there are transactions with multiple receviers and/or reccurent transactions.

The density which is the fraction of pairs of nodes connected by an edge was equal to 0,0000311.

The average degree was 2.94, in other words, on average one person has 2.94 neighbors.

The transitivity, or clustering coefficient, is the overall probability for the network to have adjacent nodes interconnected, thus revealing the existence of tightly connected communities, in our network this value was equal to 0.0012, which is close to 0.

--> The network is too heavy to plot, we will need to apply some filters to be able to plot it.

Filtrer le network

In [202]:
```
data.shape
```

Out[202]:
```
(207116, 9)
```

Study the network for different hours intervals

In [203]:
```
def filter_out_Hour(data,hourlist):
    if hourlist:
        data = data[data.Hour.isin(hourlist)]
    return data

Data1_4= filter_out_Hour(data,range(1,5))
Data5_9= filter_out_Hour(data,range(5,10))
Data10_14= filter_out_Hour(data,range(10,15))
Data15_19= filter_out_Hour(data,range(15,20))
Data20_0= filter_out_Hour(data,[20,21,22,23,0])

listdata = [data,Data1_4,Data5_9,Data10_14,Data15_19,Data20_0]
datashape = []
datanodes = []
dataedges = []
datadensity = []
dataclustering = []
dataconnectedcom = []
dataavgdeg = []
```

In [204]:
```
for df in listdata:
    g = nx.from_pandas_edgelist(df, 'Source','Destination',['Dollar
    datashape.append(df.shape)
    datanodes.append(g.number_of_nodes())
    dataedges.append(g.number_of_edges())
    datadensity.append(nx.density(g))
    dataavgdeg.append(str(mean([n for n in dict(nx.degree(g)).value
    dataconnectedcom.append(str(nx.number_connected_components(g)))
    dataclustering.append(str(nx.transitivity(g)))
```

In [205]:
```
d={'Shape':datashape , 'Nodes':datanodes, 'Edges':dataedges, 'Densi
df_sum = pd.DataFrame(d,index=['[0,24]','[24,5)','[5,10)','[10,15)'
df_sum['Avg Deg'] = (df_sum['Avg Deg'].astype('float64')).map('{:,.
df_sum['Clust'] = (df_sum['Clust'].astype('float64')).map('{:,.5f}'
df_sum.index.name = 'Hour Interval'

print(df_sum)
```

```
                      Shape  Nodes   Edges   Density  Avg Deg  Conc
Clust
Hour Interval
[0,24]          (207116, 9)  94539  139161  0.000031  2.94399  4329
0.00122
[24,5)           (42764, 9)  24036   29023  0.000100  2.41496  1187
0.00050
[5,10)           (32516, 9)  17978   23105  0.000143  2.57036  1178
0.00628
[10,15)          (43388, 9)  23890   30180  0.000106  2.52658  1576
0.00615
[15,20)          (45126, 9)  24978   33854  0.000109  2.71071  1548
0.00402
[20,24)          (43322, 9)  22338   29510  0.000118  2.64213  1512
0.00649
```

Above we can see different caracteristics (nodes, edges, density, average degree etc...) of the network for different time slopes. For the folowing analysis we have decided to focus on the transactions that occur between 5:00am and 10:00am (not included) as it represented a peak of activity.

During the interval 5am to 10am, the network has 17 978 nodes and 23 105 edges. The density (fraction of pairs of nodes connected by an edge) was at his higher value, 0.000143.

Filters to apply (focus on 5am - 10am slope)

As mentionned previously we need to apply some filters in order to reduce the size of the network.

Filter to remove transactions with the same adress as source and destination

With this filter we want to remove transactions that have the same adress as source and destination, which might be the change.

In [206]:
```
def filter_out_self_spending(data):
    return data[data["Source"]!=data["Destination"]]

Data5_9 = filter_out_self_spending(Data5_9)
Data5_9.shape
```

Out[206]:
```
(25623, 9)
```

Filter to remove adresses which belong to no cluster

We made the hypothesis taht people steal from known adresses but send to unkown addresses. Thus, we want to remove address that have a lenght superior to 20 characters but keep addresses with lenth superior to 20 characters (dst = false, because we do not want to remove unknown destination).

In [207]:
```
def filter_out_unique(data,src=True,dst=True):
    if src:
        data = data[data["Source"].str.len()<20]
    if dst:
        data = data[data["Destination"].str.len()<20]
    return data

Data5_9 = filter_out_unique(Data5_9,src=True,dst=False)
Data5_9.shape
```

Out[207]:
```
(15242, 9)
```

Filter to remove the integers which represent a group of adresses.

With this filter we want to remove all transactions that belong to a group of addresses.

In [208]:
```
def filter_out_anonym_cluster(data,src=True,dst=True):
    if src:
        data = data[~data["Source"].astype(str).str.isnumeric()]
    if dst:
        data = data[~data["Destination"].astype(str).str.isnumeric(
    return data

Data5_9 = filter_out_anonym_cluster(Data5_9,src=True,dst=True)
Data5_9.shape
```

Out[208]: (1276, 9)

Filter to keep transactions with large amount

With this filter we want to remove transactions where the amount sent is inferior to the median value (percentile = 50).

In [209]:
```
def filter_out_trasactionD_percentile(data,percentile):
    data = data[data['Dollar'] > np.percentile(data['Dollar'], perc
    return data

Data5_9 = filter_out_trasactionD_percentile(Data5_9,50)
Data5_9.shape
```

Out[209]: (638, 9)

Filter to keep recurrent recipient

With this filter we want to remove recipient that received Bitcoin from only source.

In [210]:
```
def filter_out_reccurent_recp(data):
    grp = data.groupby('Destination')['Source'].count().reset_index
    grp = grp[grp['Source']>1]
    id_list=list(grp['Destination'].unique())
    data = data[data['Destination'].isin(id_list)]
    return data

Data5_9 = filter_out_reccurent_recp(Data5_9)
Data5_9.shape
```

Out[210]: (465, 9)

Filter to remove transaction with a degree of 1

With the same idea as the previous filter, we want to remove nodes with a degree of 1.

In [211]:
```
def filter_out_self_spending(data):
    g = nx.Graph()
    g = nx.from_pandas_edgelist(data, 'Source','Destination',['Dolla
    degree_dict = dict(g.degree(g.nodes()))
    degree_dict = dict(filter(lambda elem: elem[1] != 1, degree_dict
    id_list=list(degree_dict.keys())
    data = data[data['Source'].isin(id_list)]
    return data
```

In [212]:
```
Data5_9 = filter_out_self_spending(Data5_9)
Data5_9.shape
```

Out[212]:
```
(462, 9)
```

Define a new network ((focus on 5am - 10am slope)

In [213]:
```
G = nx.Graph()
G = nx.from_pandas_edgelist(Data5_9, 'Source','Destination',["Dolla
```

Informations

In [214]:
```
n = G.number_of_nodes()
m = G.number_of_edges()
print("Number of nodes :", str(n))
print("Number of edges :", str(m))
print("Number of connected components :",str(nx.number_connected_co
print ("avg degree: "+str(mean([n for n in dict(nx.degree(G)).value
print('Density :', nx.density(G))
print('Transitivity :', nx.transitivity(G))
```

```
Number of nodes : 45
Number of edges : 145
Number of connected components : 1
avg degree: 6.444444444444445
Density : 0.14646464646464646
Transitivity : 0.35067437379576105
```

The filtered network has 45 nodes with 145 edges.

The density is equal to 0.1464, largely higher than the one obtain previously with the entire network, suggesting that there are more pairs of nodes connected between them.

The average degree is 6.44, in other words, on average one person has 6.44 neighbors, more than two times the average degree of the entire network.

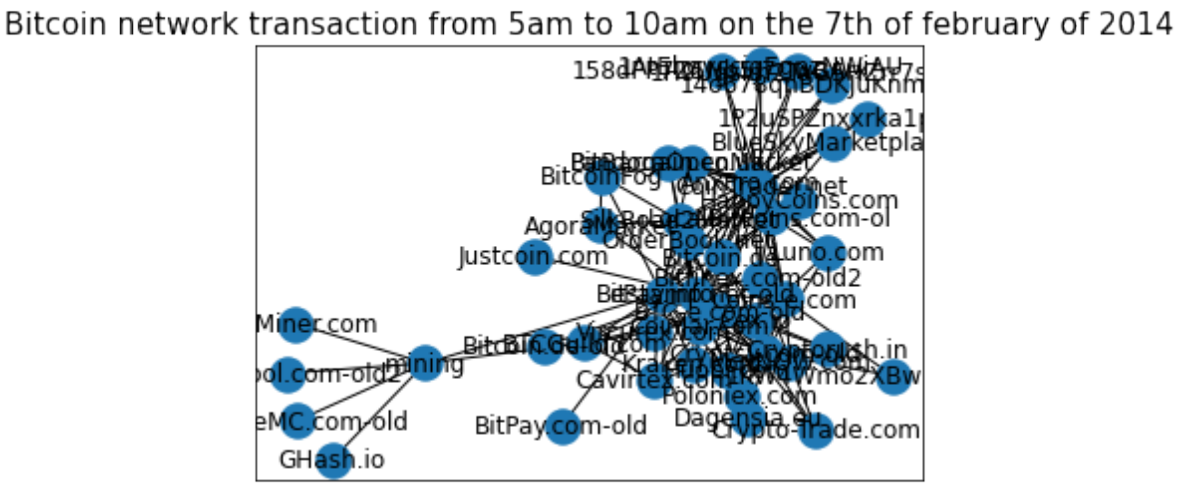
The transitivity is equal to 0.305, largely higher than the one obtain previously with the entire network.

Plot the network

The reduced network is vizualized below thanks to different methods.

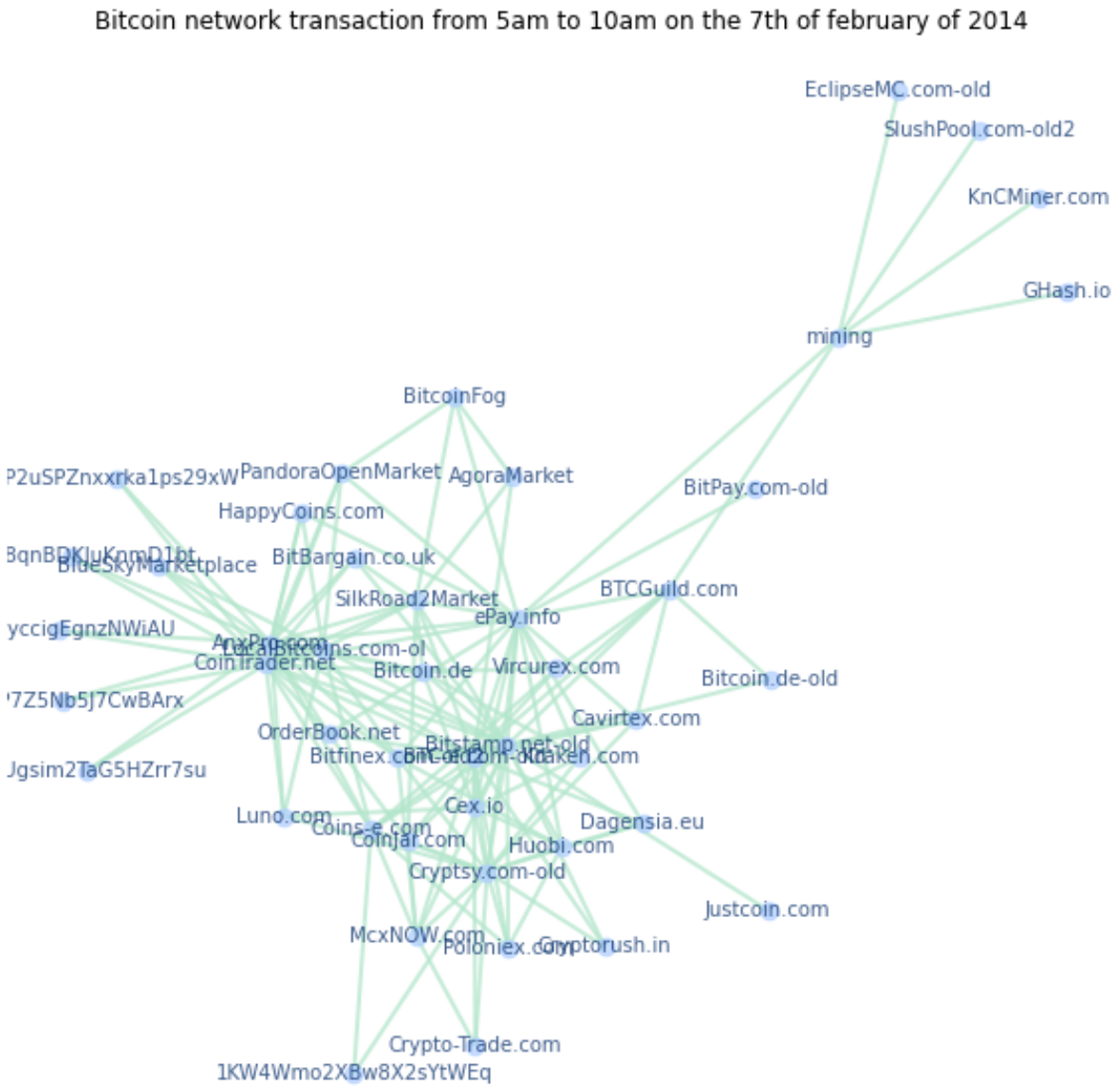
In [215]:
```
nx.draw_networkx(G)
plt.title('Bitcoin network transaction from 5am to 10am on the 7th
```

Out[215]:
```
Text(0.5, 1.0, 'Bitcoin network transaction from 5am to 10am on th
e 7th of february of 2014')
```



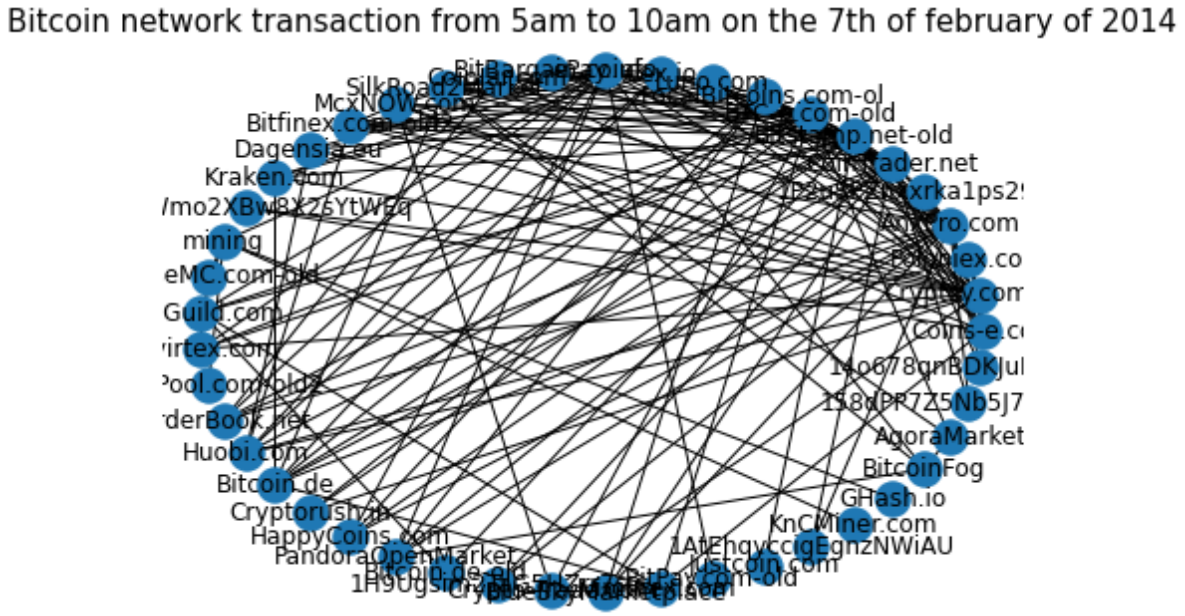
In [224]:
```
fig, ax = plt.subplots(figsize=(10, 10))
nx.draw(G,
        with_labels=True,
        ax=ax,
        node_size=75,
        node_color='#b3d1ff',
        edge_color='#b3e6cc',
        width =2.0,
        stype= 'dashed',
        font_size=10.0,
        font_color='#002966',
        alpha=0.75)
ax.set_title('Bitcoin network transaction from 5am to 10am on the 7
```

Out[224]:
```
Text(0.5, 1.0, 'Bitcoin network transaction from 5am to 10am on th
e 7th of february of 2014')
```



In [216]:
```
nx.draw_circular(G, with_labels=True)
plt.title('Bitcoin network transaction from 5am to 10am on the 7th
```

Out[216]:
```
Text(0.5, 1.0, 'Bitcoin network transaction from 5am to 10am on th
e 7th of february of 2014')
```



Network Influencers

Degree Centrality

Degree centrality is a measure of the number of connections a particular node has in the network. It is based on the fact that important nodes have many connections.

In [217]:
```
degree_centrality = nx.degree_centrality(G)
dc = sorted(degree_centrality, key=degree_centrality.get, reverse=T
dc
```

Out[217]:
```
['Bitstamp.net-old',
 'AnxPro.com',
 'CoinTrader.net',
 'BTC-e.com-old',
 'ePay.info']
```

Eigenvector Centrality

It not just assess how many addresses one is connected too, but the type of addresses one is connected with that can decide the importance of a node. It decides that a node is important if it is connected to other important nodes.

In [218]:
```
eigenvector_centrality = nx.eigenvector_centrality(G)
evc = sorted(eigenvector_centrality, key=eigenvector_centrality.get
evc
```

Out[218]:
```
['Bitstamp.net-old',
 'BTC-e.com-old',
 'ePay.info',
 'AnxPro.com',
 'CoinTrader.net']
```

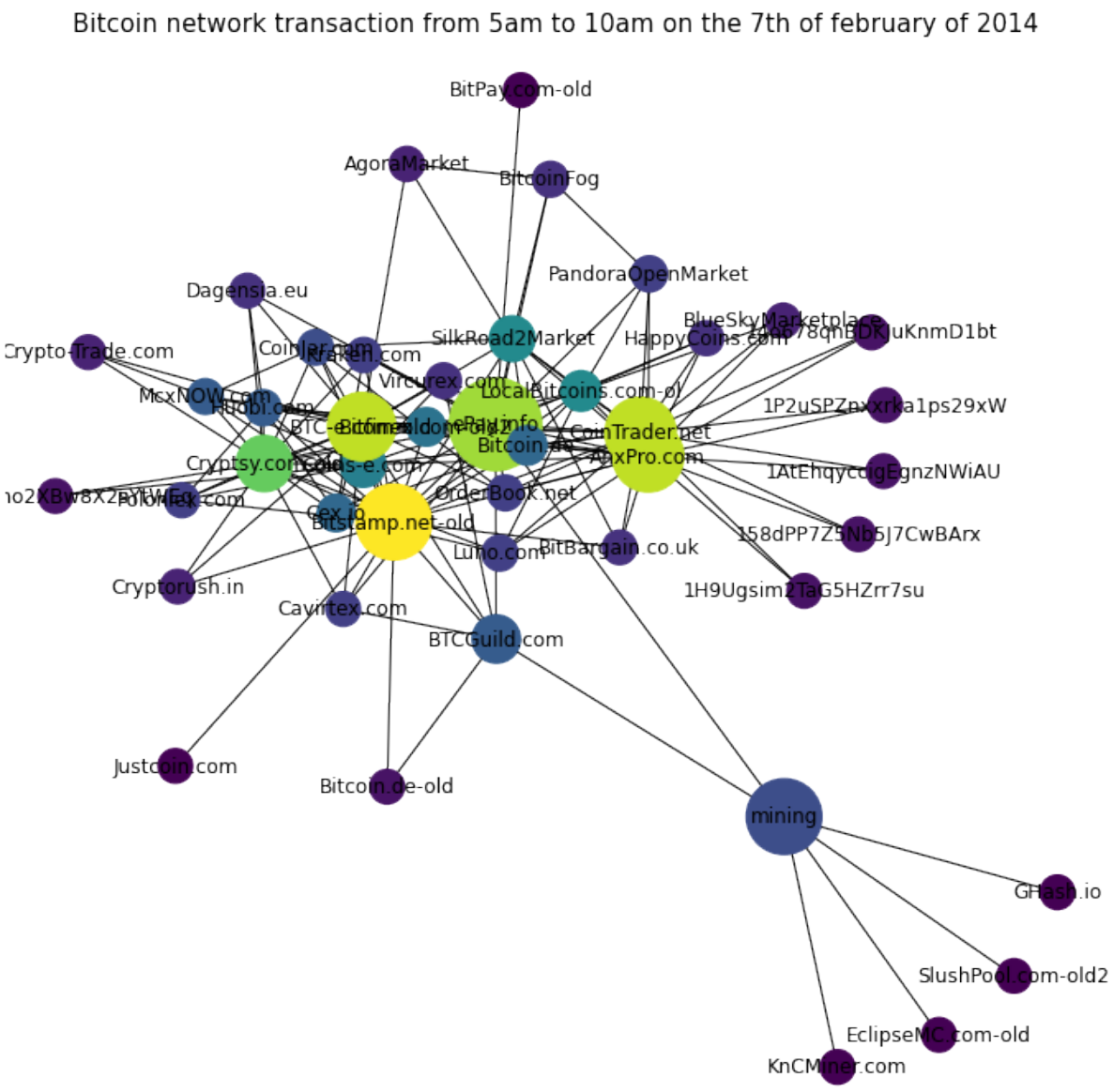
Betweenness Centrality

The Betweenness Centrality quantifies how many times a particular node comes in the shortest chosen path between two other nodes. The nodes with high betweenness centrality play a significant role in the communication/information flow within the network. The nodes with high betweennesss centrality can have strategic control and influence on others.

Visualization of the network such that the node color varies with "Degree" and node size with "Betweenness Centrality".

In [219]:
```
pos = nx.spring_layout(G)
betCent = nx.betweenness_centrality(G, normalized=True, endpoints=T
node_color = [20000.0 * G.degree(v) for v in G]
node_size =  [v * 10000 for v in betCent.values()]
plt.figure(figsize=(12,12))
nx.draw_networkx(G, pos=pos, with_labels=True,
                 node_color=node_color,
                 node_size=node_size)
plt.axis('off')
plt.title('Bitcoin network transaction from 5am to 10am on the 7th
```

Out[219]:
```
Text(0.5, 1.0, 'Bitcoin network transaction from 5am to 10am on th
e 7th of february of 2014')
```



We obtain the following labels of nodes with the highest betweenness centrality.

In [220]:
```
bc = sorted(betCent, key=betCent.get, reverse=True)[:5]
bc
```

Out[220]:
```
['ePay.info', 'Bitstamp.net-old', 'mining', 'AnxPro.com', 'CoinTra
der.net']
```

Conclusions : Degree centrality, Eigenvector centrality, Betweenness centrality

In [221]:
```
total = pd.DataFrame({'Degree centrality':dc, 'Eigenvector centrali
total
```

Out[221]:

	Degree centrality	Eigenvector centrality	Betweenness centrality
0	Bitstamp.net-old	Bitstamp.net-old	ePay.info
1	AnxPro.com	BTC-e.com-old	Bitstamp.net-old
2	CoinTrader.net	ePay.info	mining
3	BTC-e.com-old	AnxPro.com	AnxPro.com
4	ePay.info	CoinTrader.net	CoinTrader.net

We can see that some nodes are common between Degree Centrality, which is a measure of degree, and Betweenness Centrality which controls the information flow. It is natural that nodes that are more connected also lie on shortest paths between other nodes. The node "Bitstamp.net-old" is an important node as it is crucial according to all three centrality measures that we had considered.

Network Structures

In network theory, a "clique" is essentially defined on the social version of a clique : a set of nodes (addresses) that are completely connected by an edge to every other node in the set. It is then, a completely connected graph.

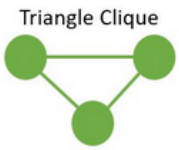
Simplest clique : an edge is the simplest clique possible

Simplest complex clique : 3 nodes fully connected : a triangle

Please do not run !

In [135]:
```
from IPython.display import Image
PATH = "/Users/Louis/Downloads/"
Image(filename = PATH + "Triangle.png", width=100, height=100)
```

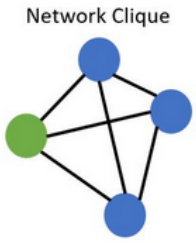
Out[135]:



Please do not run !

In [136]:
```
Image(filename = PATH + "Capture.png", width=100, height=100)
```

Out[136]:



Triangles

Triangles correspond to the simplest complex clique.

Identify triangle relationships

In the Bitcoin Transactions Network, each node has an associated address label. One potential application of triangle-finding algorithms is to find out whether a particular type of address is more likely to be in a triangle with one another.

In [137]:
```
from itertools import combinations

# Define is_in_triangle()
def is_in_triangle(G, n):
    """
    Checks whether a node `n` in graph `G` is in a triangle relatio

    Returns a boolean.
    """
    in_triangle = False

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n), 2):

        # Check if an edge exists between n1 and n2
        if G.has_edge(n1, n2):
            in_triangle = True
            break
    return in_triangle

is_in_triangle(G, "Bitstamp.net-old")
```

Out[137]:
```
True
```

In [138]:
```
for node in sorted(list(G.nodes())[:10]):
    x = is_in_triangle(G, node)
    if x == True:
        print(f'{node}: {x}')
```

```
1P2uSPZnxxrka1ps29xW: True
AnxPro.com: True
BTC-e.com-old: True
Bitstamp.net-old: True
CoinTrader.net: True
Coins-e.com: True
Cryptsy.com-old: True
LocalBitcoins.com-ol: True
Luno.com: True
Poloniex.com: True
```

Find nodes involved in triangles

We can then use a function to extract all of the nodes involved in a triangle relationship with a given node (address).

In [139]:
```
# Write a function that identifies all nodes in a triangle relation
def nodes_in_triangle(G, n):
    """
    Returns the nodes in a graph `G` that are involved in a triangl
    """
    triangle_nodes = set([n])

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n), 2):

        # Check if n1 and n2 have an edge between them
        if G.has_edge(n1, n2):

            # Add n1 to triangle_nodes
            triangle_nodes.add(n1)

            # Add n2 to triangle_nodes
            triangle_nodes.add(n2)
    return triangle_nodes

nodes_in_triangle(G, "AgoraMarket")
```

Out[139]:
```
{'AgoraMarket', 'BTC-e.com-old', 'BitcoinFog', 'SilkRoad2Market'}
```

Find open triangles

Identify whether a node is present in an open triangle with its neighbors.

In [140]:
```
# Define node_in_open_triangle()
def node_in_open_triangle(G, n):
    """
    Checks whether pairs of neighbors of node `n` in graph `G` are
    """
    in_open_triangle = False

    # Iterate over all possible triangle relationship combinations
    for n1, n2 in combinations(G.neighbors(n), 2):

        # Check if n1 and n2 do NOT have an edge between them
        if not G.has_edge(n1, n2):

            in_open_triangle = True

            break

    return in_open_triangle

# Compute the number of open triangles in T
num_open_triangles = 0

# Iterate over all the nodes in T
for n in G.nodes():

    # Check if the current node is in an open triangle
    if node_in_open_triangle(G, n):

        # Increment num_open_triangles
        num_open_triangles += 1

print(f'{num_open_triangles} nodes in graph G are in open triangles
```

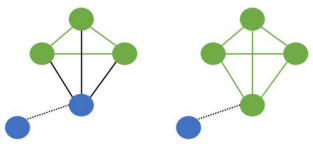
```
28 nodes in graph G are in open triangles.
```

Maximal Cliques

Maximal cliques correspond to clique that cannot be extended by adding another node in the graph.

In [141]:
```
Image(filename = PATH + "Max_ex.png", width=150, height=150)
```

Out[141]:



For example, the sub-clique of the 3 green nodes can be extended by one blue node to form a large clique.

As such, these 3 green nodes do not form a maximal clique in the graph.

The 4 nodes connected as a clique together cannot be extended and still remain a clique, as the remaining node is not fully connected to the other four nodes.

As such, these 4 nodes constitute a maximal clique.

NetworkX provides a function that allows you to identify the nodes involved in each maximal clique in a graph: nx.find_cliques(G).

In [142]:
```
# Define maximal_cliques()
def maximal_cliques(G, n):
    """
    Finds all maximal cliques in graph `G` that are of size `size`.
    """
    mcs = []
    for clique in nx.find_cliques(G):
        if len(clique) == n:
            mcs.append(clique)
    return mcs

maximal_cliques(G, 6)
```

Out[142]:
```
[['Bitstamp.net-old',
  'Bitcoin.de',
  'SilkRoad2Market',
  'AnxPro.com',
  'CoinTrader.net',
  'LocalBitcoins.com-ol'],
 ['Bitstamp.net-old',
  'BTC-e.com-old',
  'SilkRoad2Market',
  'AnxPro.com',
  'CoinTrader.net',
  'ePay.info'],
 ['Bitstamp.net-old',
  'BTC-e.com-old',
  'CoinTrader.net',
  'AnxPro.com',
  'Bitfinex.com-old2',
  'ePay.info']]
```

Finding Cliques

Cliques are "groups of nodes that are fully connected to one another", while a maximal clique is a clique that cannot be extended by adding another node in the graph.

In [143]:
```
cliques = sorted([len(cl) for cl in nx.find_cliques(G)], reverse=Tr
print(cliques)
print(f'There are {len(cliques)} cliques.')
```

```
[6, 6, 6, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 4, 4, 4,
4, 4, 4, 4, 4, 4, 4, 4, 4, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
3, 3, 3, 3, 2, 2, 2, 2, 2, 2, 2]
There are 55 cliques.
```

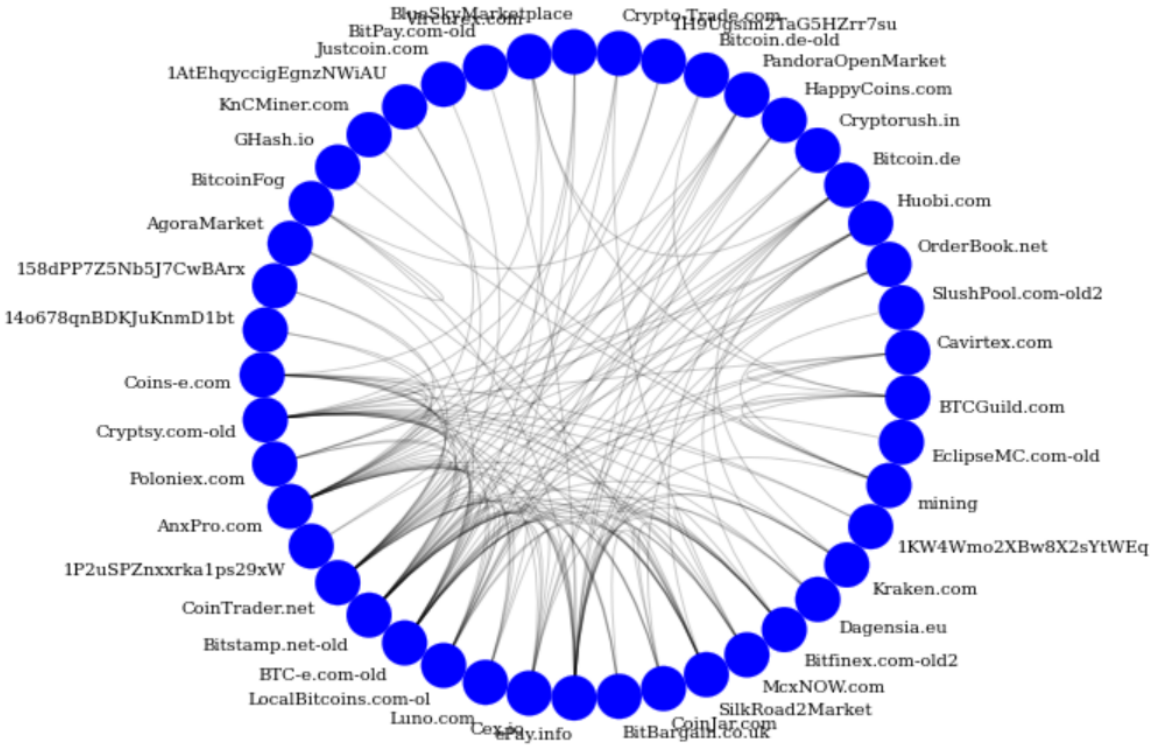
Finding a particular maximal clique, and then plotting that clique.

Addresses that are part of the largest maximal clique, and plot the subgraph of that/one of those clique(s) using a CircosPlot.

In [145]:
```
Image(filename = PATH + "Maximal.png", width=1000, height=1000)
```

Out[145]:

```
The largest maximal clique consists of 45 addresses
```



Finding important addresses

Look at important nodes using of the degree_centrality() and betweenness_centrality() functions in NetworkX to compute each of the respective centrality scores, and then use that information to find the "important nodes".

In other words, here we investigate the addresses that have collaborated with the most number of addresses.

In [146]:
```
# Compute the degree centralities of G: deg_cent
deg_cent = nx.degree_centrality(G)

# Sorting the dictionary
deg_cent = {k: v for k, v in sorted(deg_cent.items(), key=lambda it

# Compute the maximum degree centrality: max_dc
max_dc = max(list(deg_cent.values()))

# Find the user(s) that have collaborated the most: prolific_addres
prolific_address = [n for n, dc in deg_cent.items() if dc == max_dc

# Print the most prolific address
print(f'The most prolific address: {prolific_address}')
```

```
The most prolific address: ['Bitstamp.net-old']
```

In [147]:
```
# Maximal degree centrality
max_dc
```

Out[147]:
```
0.5
```

In [148]:
```
# Degree centrality of the different addresses
deg_cent
```

Out[148]:
```
{'Bitstamp.net-old': 0.5,
 'AnxPro.com': 0.4545454545454546,
 'CoinTrader.net': 0.4545454545454546,
 'BTC-e.com-old': 0.4545454545454546,
 'ePay.info': 0.4318181818181818,
 'Cryptsy.com-old': 0.38636363636363635,
 'LocalBitcoins.com-ol': 0.25,
 'SilkRoad2Market': 0.25,
 'Coins-e.com': 0.2272727272727273,
 'Bitfinex.com-old2': 0.20454545454545456,
 'Cex.io': 0.18181818181818182,
 'Bitcoin.de': 0.18181818181818182,
 'McxNOW.com': 0.1590909090909091,
 'BTCGuild.com': 0.1590909090909091,
 'Huobi.com': 0.1590909090909091,
 'CoinJar.com': 0.13636363636363635,
 'mining': 0.13636363636363635,
 'Poloniex.com': 0.11363636363636365,
 'Luno.com': 0.11363636363636365,
```

Community

Compute communities

In [149]:
```
coms = algorithms.louvain(G, weight='weight', resolution=1., random
coms.to_node_community_map()
```

Out[149]:
```
defaultdict(list,
            {'Coins-e.com': [0],
             'Cryptsy.com-old': [0],
             'Poloniex.com': [0],
             'Bitstamp.net-old': [0],
             'BTC-e.com-old': [0],
             'CoinJar.com': [0],
             'McxNOW.com': [0],
             'Bitfinex.com-old2': [0],
             'Dagensia.eu': [0],
             '1KW4Wmo2XBw8X2sYtWEq': [0],
             'BTCGuild.com': [0],
             'Cavirtex.com': [0],
             'OrderBook.net': [0],
             'Huobi.com': [0],
             'Cryptorush.in': [0],
             'Bitcoin.de-old': [0],
             'Crypto-Trade.com': [0],
             'Vircurex.com': [0],
             'Justcoin.com': [0],
             'AnxPro.com': [1],
             '1P2uSPZnxxrka1ps29xW': [1],
             'CoinTrader.net': [1],
             'LocalBitcoins.com-ol': [1],
             'Luno.com': [1],
             'BitBargain.co.uk': [1],
             'HappyCoins.com': [1],
             'PandoraOpenMarket': [1],
             '1H9Ugsim2TaG5HZrr7su': [1],
             'BlueSkyMarketplace': [1],
             '1AtEhqyccigEgnzNWiAU': [1],
             '158dPP7Z5Nb5J7CwBArx': [1],
             '14o678qnBDKJuKnmD1bt': [1],
             'Cex.io': [2],
             'ePay.info': [2],
             'SilkRoad2Market': [2],
             'Kraken.com': [2],
             'Bitcoin.de': [2],
             'BitPay.com-old': [2],
             'BitcoinFog': [2],
             'AgoraMarket': [2],
             'mining': [3],
             'EclipseMC.com-old': [3],
             'SlushPool.com-old2': [3],
             'KnCMiner.com': [3],
             'GHash.io': [3]})
```

Set node attributes

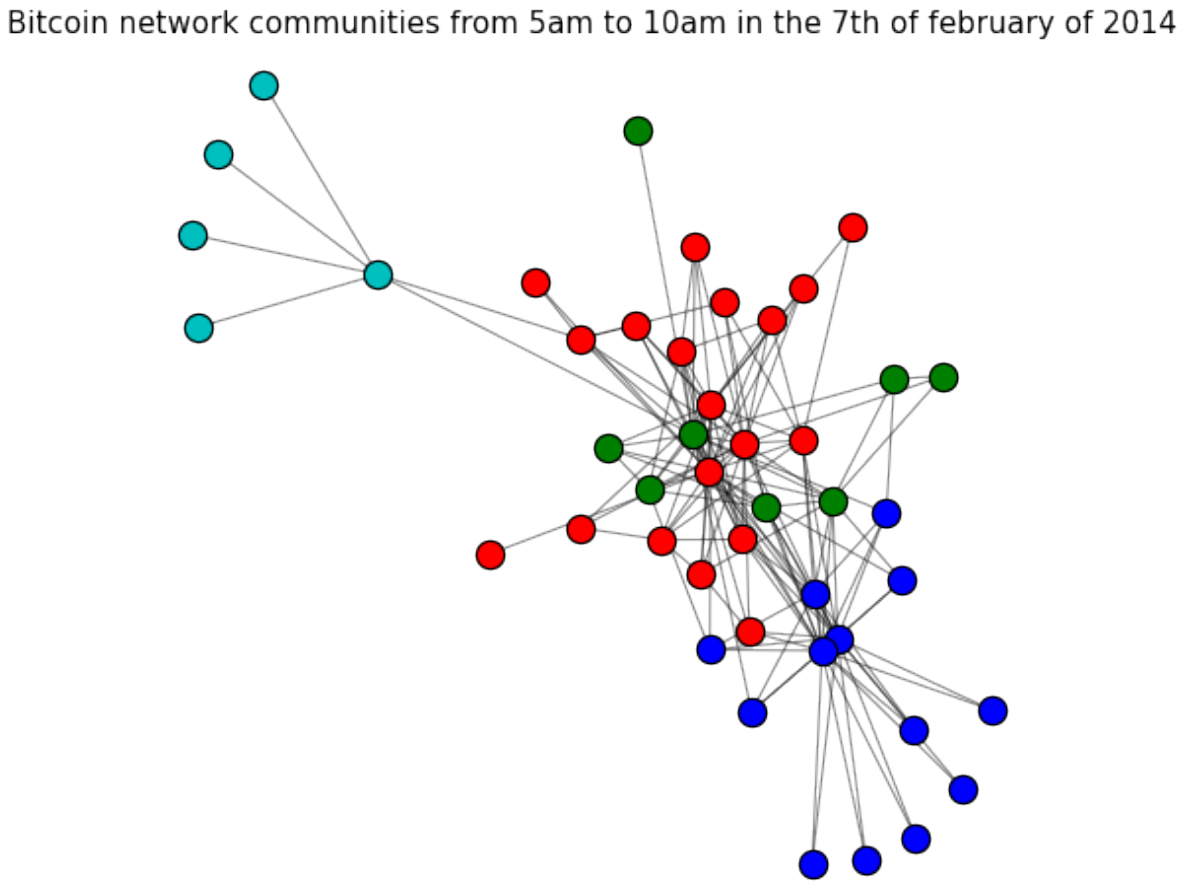
In [150]:
```
nx.set_node_attributes(G, coms, "Community")
```

Plot the communities

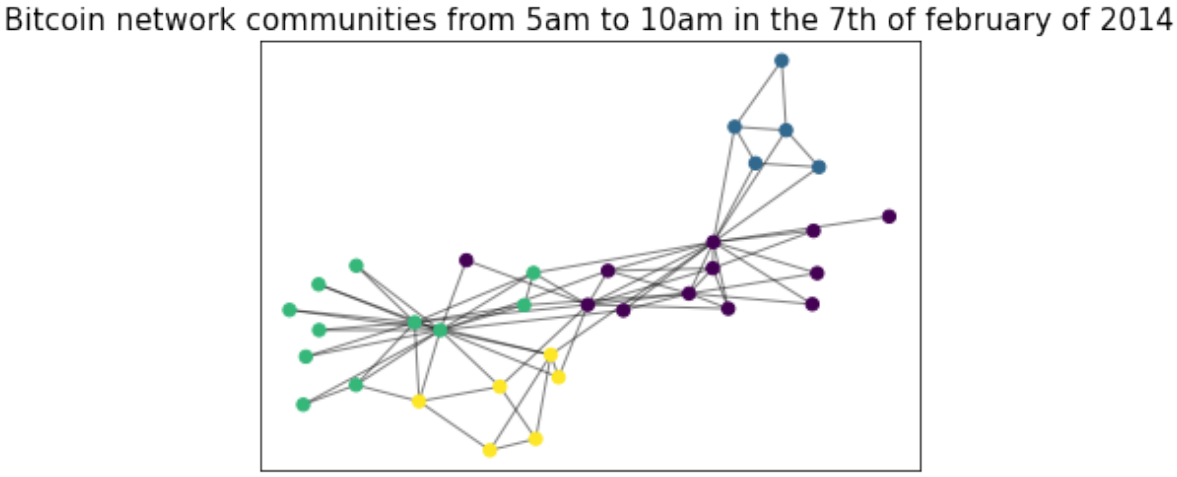
On the following graph, each color represents a different community in the network.

In [151]:
```
viz.plot_network_clusters(G, coms)
plt.title('Bitcoin network communities from 5am to 10am in the 7th
```

Out[151]:
```
Text(0.5, 1.0, 'Bitcoin network communities from 5am to 10am in th
e 7th of february of 2014')
```



In [236]:
```
import community as community_louvain
import matplotlib.cm as cm

G = nx.karate_club_graph()
# compute the best partition
partition = community_louvain.best_partition(G)

# draw the graph
pos = nx.spring_layout(G)
# color the nodes according to their partition
cmap = cm.get_cmap('viridis', max(partition.values()) + 1)
nx.draw_networkx_nodes(G, pos, partition.keys(), node_size=40,
cmap=cmap, node_color=list(partition.values()))
nx.draw_networkx_edges(G, pos, alpha=0.5)
plt.title('Bitcoin network communities from 5am to 10am in the 7th
plt.show()
```



In [152]:
```
#from nxviz import CircosPlot
#c = CircosPlot(G, node_color='coms', node_grouping='coms')
#c.draw()
```

Betweeness & Page Rank

Betweeness Centrality

Betweenness is a measure of centrality based on shortest paths. The betweenness centrality for each node is the number of these shortest paths that pass through the node.

In [153]:
```
betweenness_dict = nx.betweenness_centrality(G)
nx.set_node_attributes(G, betweenness_dict, 'betweenness')
```

Page Rank

PageRank is an algorithm used by Google Search to rank web pages in their search engine results. It is a way of measuring the importance of website pages. PageRank works by counting the number and quality of links to a page to determine a rough estimate of how important the website is.

In [154]:
```
pagerank_dict = nx.pagerank(G)
nx.set_node_attributes(G, pagerank_dict, 'pagerank')
```

Top 5

Below are the five most important nodes regarding their betweeness measure, suggesting that they are at the heart of the network.

In [155]:
```
sorted_betweenness = sorted(betweenness_dict.items(), key=itemgette

degree_dict = dict(G.degree(G.nodes()))
degree_dict = dict(filter(lambda elem: elem[1] != 1, degree_dict.it
```

In [156]:
```
top_betweenness = sorted_betweenness[:5]

for tb in top_betweenness:
    degree = degree_dict[tb[0]]
    pagerank = pagerank_dict[tb[0]]
    print("Name:", tb[0], "| Betweenness Centrality:", round(tb[1],
```

```
Name: ePay.info | Betweenness Centrality: 0.285989 | Degree: 19 |
Page Rank: 0.061687
Name: Bitstamp.net-old | Betweenness Centrality: 0.177125 | Degree
: 22 | Page Rank: 0.067274
Name: mining | Betweenness Centrality: 0.175476 | Degree: 6 | Page
Rank: 0.039888
Name: AnxPro.com | Betweenness Centrality: 0.151271 | Degree: 20 |
Page Rank: 0.063359
Name: CoinTrader.net | Betweenness Centrality: 0.151271 | Degree:
20 | Page Rank: 0.063359
```

Shortest path

Pathfinding algorithms are important because they provide another way of assessing node importance.

To compute the average shortest path, we need to first compute the largest connected component, otherwise the distance is infinite between some nodes.

In [157]:
```
cc = G.subgraph(sorted(nx.connected_components(G), key=len, reverse
print ("average shortest parth: ",str(nx.average_shortest_path_leng
print( 'Diameter :', nx.diameter(cc))
```

```
average shortest parth:  2.294949494949495
Diameter : 4
```

On average, the average number of steps along the shortest paths for all possible pairs of network nodes is 2.29. Diameter is the maximum distance between any pair of nodes, here it is 4.

In [158]:
```
print("nodes of highest degree:")
sorted(nx.degree(G), key=lambda x: x[1],  reverse=True)[:10]
```

```
nodes of highest degree:
```

Out[158]:
```
[('Bitstamp.net-old', 22),
 ('AnxPro.com', 20),
 ('CoinTrader.net', 20),
 ('BTC-e.com-old', 20),
 ('ePay.info', 19),
 ('Cryptsy.com-old', 17),
 ('LocalBitcoins.com-ol', 11),
 ('SilkRoad2Market', 11),
 ('Coins-e.com', 10),
 ('Bitfinex.com-old2', 9)]
```

We can see that the highest degree measure is achieved for "Bitstamp.net-old", this address is connected to 22 other nodes.

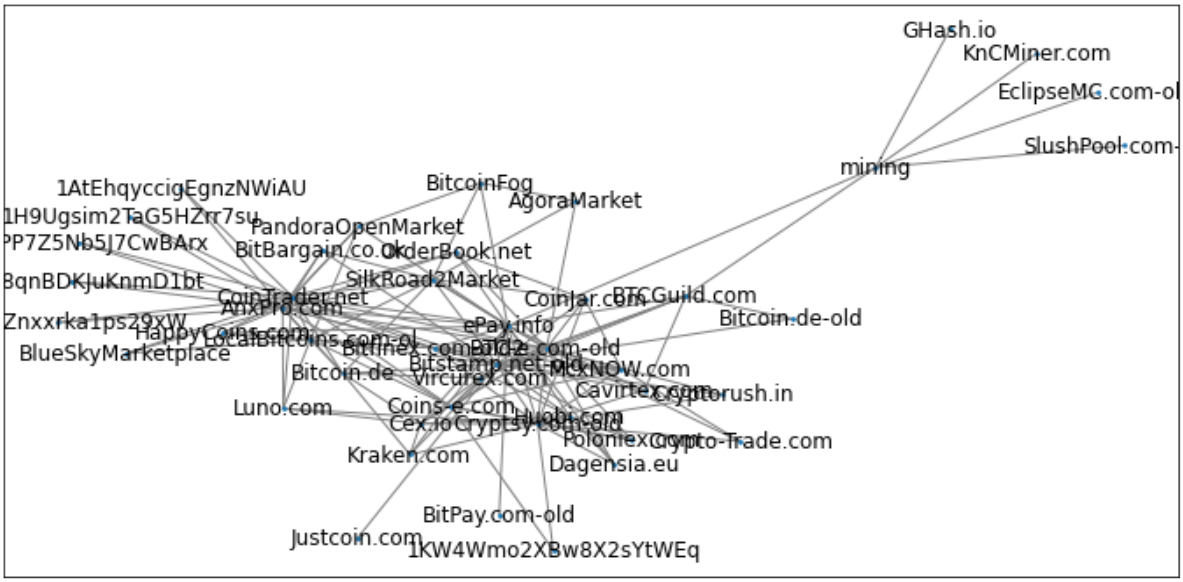
In [159]:
```
print("nodes of highest PageRank:")
sorted(nx.pagerank(G), key=lambda x: x[1],  reverse=True)[:10]
```

```
nodes of highest PageRank:
```

Out[159]:
```
['Luno.com',
 'Huobi.com',
 'Justcoin.com',
 'Cryptsy.com-old',
 'Kraken.com',
 'OrderBook.net',
 'Cryptorush.in',
 'Crypto-Trade.com',
 'Coins-e.com',
 'Poloniex.com']
```

This would be the most popular addresses based on Page Rank algorithm.

In [160]:
```
#Drawing with a classic force derected layout
plt.figure(1,figsize=(12,6))
nx.draw_networkx(G, with_labels=True,node_size=3,edge_color="grey")
```



Explore sub graphs : neighbours

Top 5 destinations by total amounts

In [161]:
```
top5 = Data5_9.groupby('Destination')['Dollar'].sum().reset_index()
top5.sort_values(by='Dollar',ascending=False).head()
```

Out[161]:

	Destination	Dollar
7	BTC-e.com-old	406379.403938
24	GHash.io	250194.002213
26	Huobi.com	198956.087906
10	BitPay.com-old	172733.784308
15	Bitstamp.net-old	136824.412817

Nodes with at least 10 neighbors

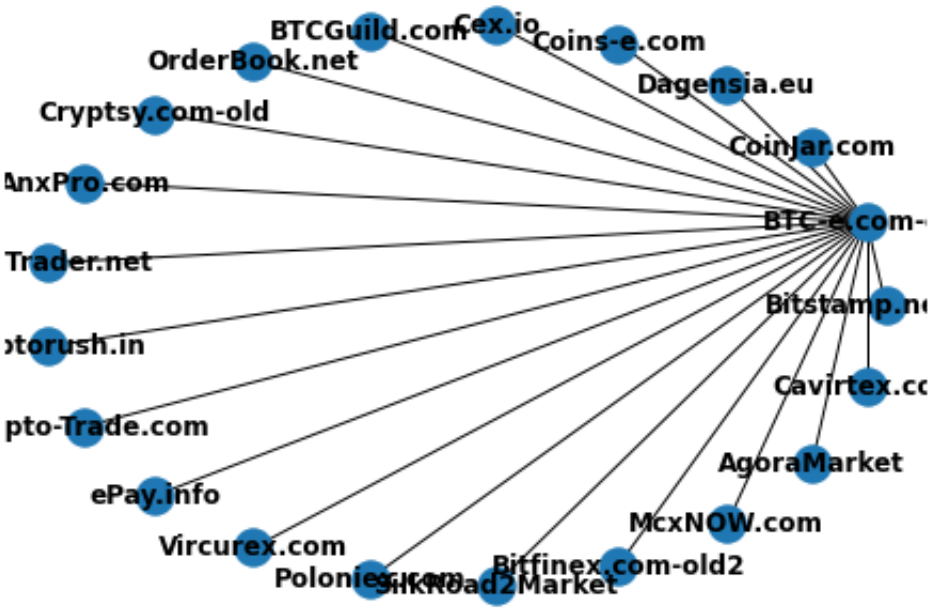
In [162]:
```
def nodes_with_m_nbrs(G,m):
    nodes = set()
    for n in G:
        if len(list(G.neighbors(n))) > m:
            nodes.add(n)
    return nodes

nodes_list = nodes_with_m_nbrs(G,10)
print(nodes_list)
```

```
{'ePay.info', 'SilkRoad2Market', 'CoinTrader.net', 'Bitstamp.net-o
ld', 'AnxPro.com', 'Cryptsy.com-old', 'LocalBitcoins.com-ol', 'BTC
-e.com-old'}
```

BTC-e.com-old

In [163]:
```
G_btc_old = nx.Graph()
G_btc_old = nx.from_pandas_edgelist(Data5_9[Data5_9['Destination']=
```

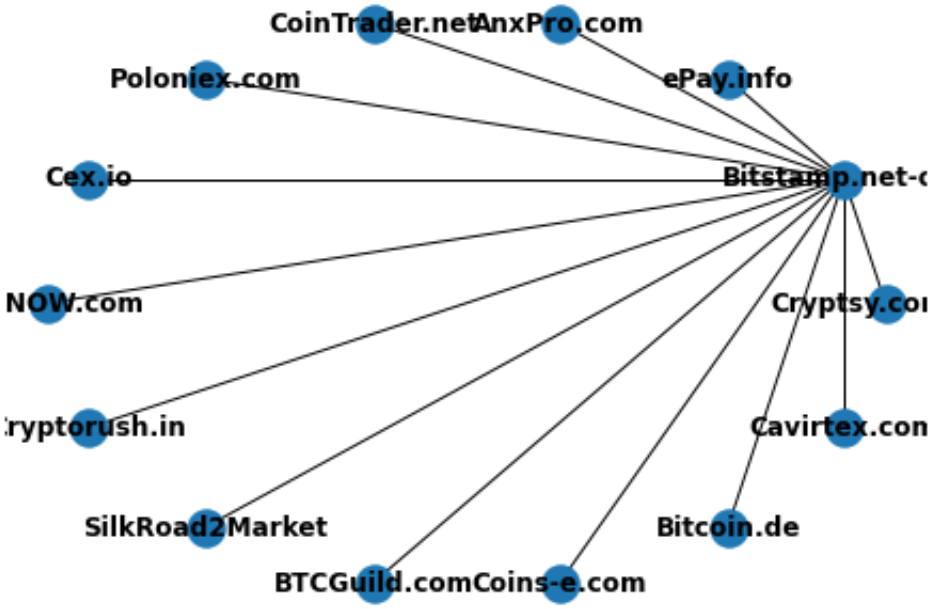
In [164]:
```
nx.draw_circular(G_btc_old, with_labels=True, font_weight='bold')
```



Study Bitstamp.net-old

In [165]:
```
G_bitstamp = nx.Graph()
G_bitstamp = nx.from_pandas_edgelist(Data5_9[Data5_9['Destination']
```

In [166]:
```
nx.draw_circular(G_bitstamp, with_labels=True, font_weight='bold')
```

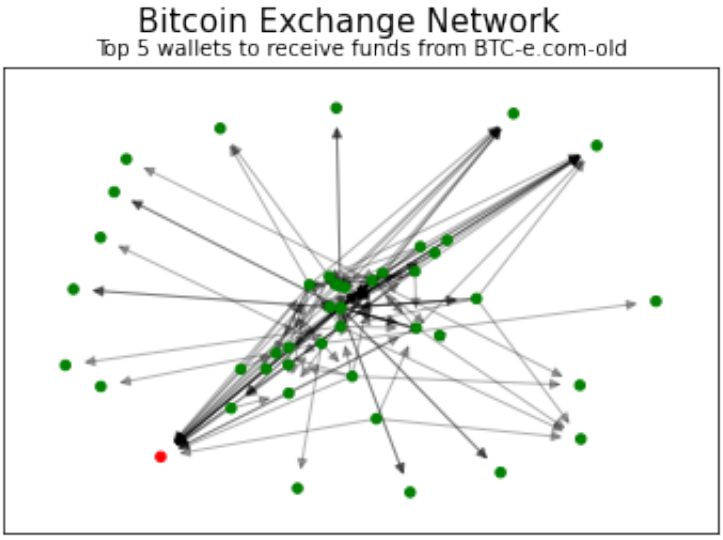


Visualize network of transaction

For the reduced network

In [167]:
```
nx_plot = pd.Series.to_frame(Data5_9.groupby(['Source', 'Destinatio
G1 = nx.from_pandas_edgelist(nx_plot[0:200], 'Source', 'Destination

color_list = []
for node in G1.nodes():
    if node == 'BTC-e.com-old':
        color_list.append('red')
    else:
        color_list.append('green')

pos = nx.spring_layout(G1)
nx.draw_networkx_nodes(G1,pos,node_color=color_list,node_size=20)
nx.draw_networkx_edges(G1, pos, alpha=0.3)
plt.title('Top 5 wallets to receive funds from BTC-e.com-old', font
plt.suptitle('Bitcoin Exchange Network', fontsize=15)
```

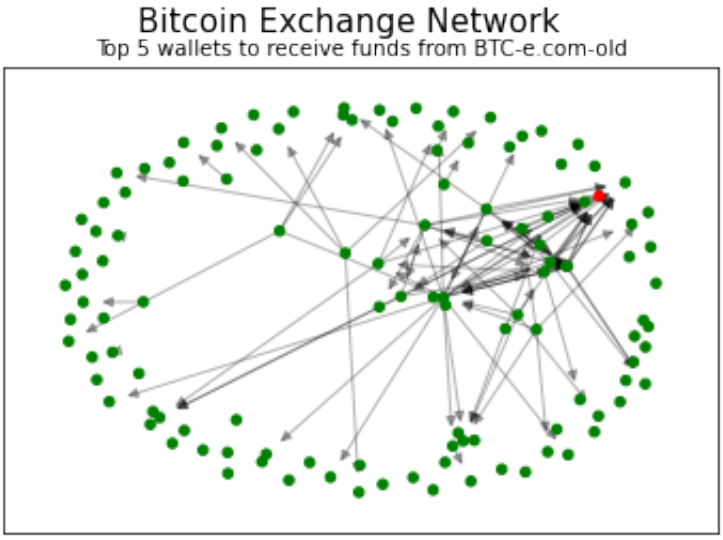
Out[167]:
```
Text(0.5, 0.98, 'Bitcoin Exchange Network')
```



For the entire network

In [168]:
```
nx_plot = pd.Series.to_frame(data.groupby(['Source', 'Destination']
G2 = nx.from_pandas_edgelist(nx_plot[0:200], 'Source', 'Destination

color_list = []
for node in G2.nodes():
    if node == 'BTC-e.com-old':
        color_list.append('red')
    else:
        color_list.append('green')

pos = nx.spring_layout(G2)
nx.draw_networkx_nodes(G2,pos,node_color=color_list,node_size=20)
nx.draw_networkx_edges(G2, pos, alpha=0.3)
plt.title('Top 5 wallets to receive funds from BTC-e.com-old', font
plt.suptitle('Bitcoin Exchange Network', fontsize=15)
```

Out[168]:
```
Text(0.5, 0.98, 'Bitcoin Exchange Network')
```



GEPHI graph

In [3]:
```
from IPython.display import Image
from IPython.core.display import HTML
```

Export to excel

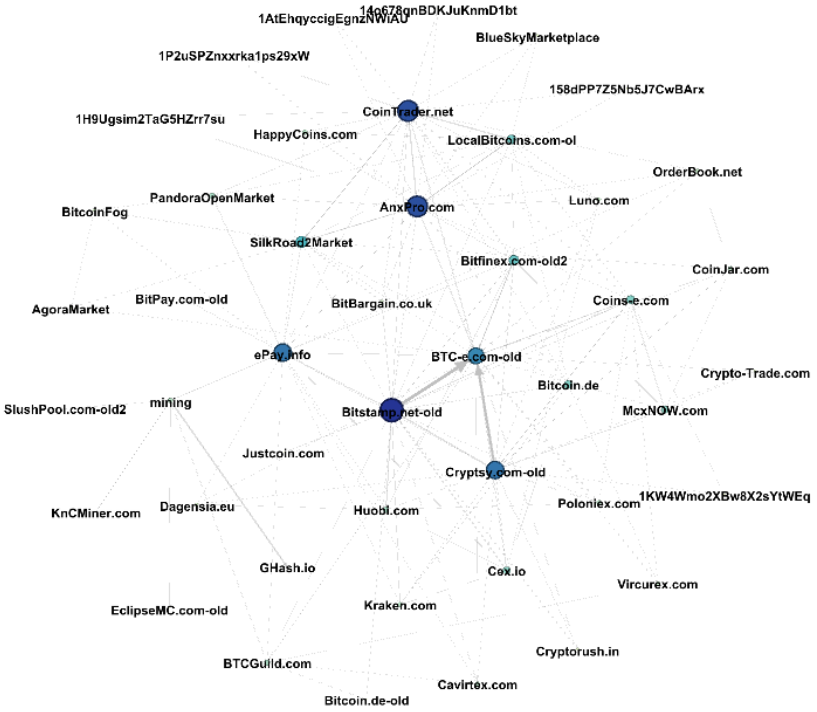
In [5]:
```
# Data5_9.to_csv('for_gephi.csv')
```

Import image from gephi

Please do not run !

In [9]:
```
PATH = "/Users/Daudenthun/Documents/M2/Network analysis/Projet/"
Image(filename = PATH + "degree_gephi.png", width=1000, height=1000
```

Out[9]:

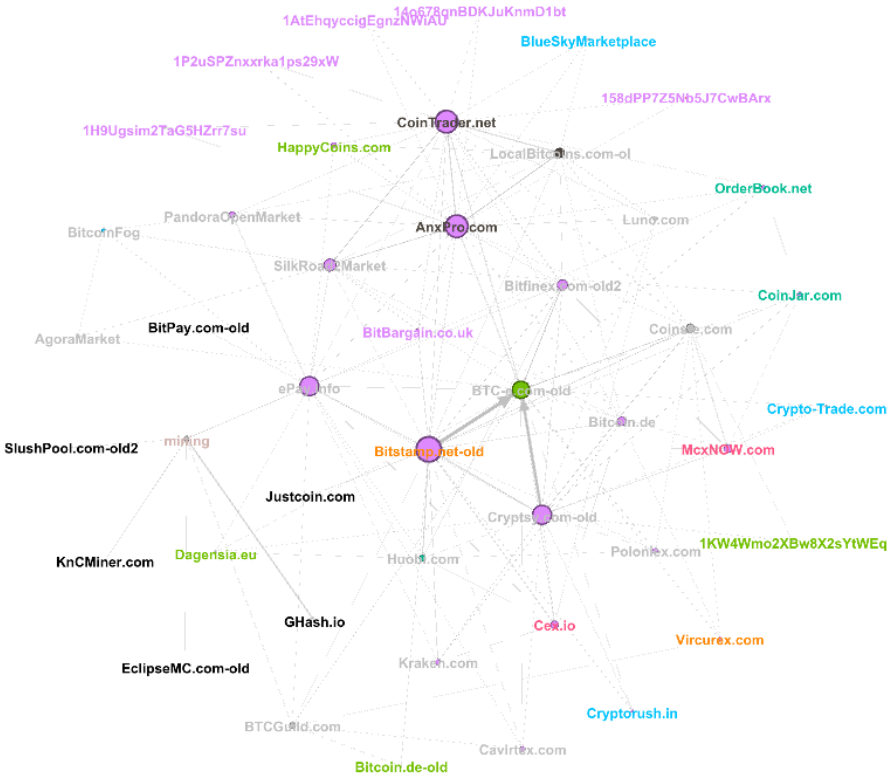


The graph above represents the reduced network. Nodes are highlithed regarding their degree. The node with the higher degree is Bitstamp.net-old.

Please do not run !

In [10]:
```
Image(filename = PATH + "connected_cluster_gephi.png", width=1000,
```

Out[10]:



The graph above represents the reduced network. Nodes are bigger regarding their degree, the color is then determined regarding their connected component. An address is sorted regarding its cluster coefficient, each color represent a different cluster.

Graph Learning

Link Prediction

With Link Prediction, given a graph G, we aim to predict new edges. Predictions are useful to predict future relations or missing edges when the graph is not fully observed for example, or when new addresses join the network.

Link prediction for a new address would simple be a suggestion of a destination : addresses pairs.

In link prediction, we simply try to build a "similarity measure" between pairs of nodes and link the most similar nodes.

The question is consequently to identify and compute the right similarity scores.

Adamic-Adar index : for each common neighbor of nodes "i" and "j", we add 1 divided by the total number of neighbors of that node. The concept is that common elements with very large neighborhoods are less significant when predicting a connection between two nodes compared to elements shared between a small number of nodes. AA Intuition:

- A common node with ONLY them in common is worth the most
- A common node connected to everyone is worth the less
- The higher the size of its neighborhood, the lesser its value

Preferential attachment : every time a node join the network, it creates a link with nodes with probability proportional to their degrees. Score not based on common neighbors. PA Intuition:

- Assign different scores to nodes at network distance > 2
- Two nodes with many neighbors more likely to have new ones than nodes with few neighbors

In [173]:
```
import random

g = G

#We first sample existing edges as training examples
training_sample_size=int(g.number_of_edges()/3)
all_edges = {frozenset((u,v)) for (u,v) in g.edges}
training_edges = random.sample(list(all_edges),training_sample_size

#We then sample an equal number of pairs of nodes without edges
all_node_pairs= {frozenset((u,v)) for u in g.nodes for v in g.nodes
non_edges=all_node_pairs-all_edges
training_non_edges= random.sample(list(non_edges),training_sample_s

print("nb edges for training: ",len(training_edges))
print("nb non-edges for training: ",len(training_non_edges))
```

```
nb edges for training:  48
nb non-edges for training:  48
```

In [174]:
```
#We remove edges in the training set from the original graph
training_graph = g.copy()
training_graph.remove_edges_from([(u,v) for u,v in training_edges])

#And then compute some heuristics, for the sake of example, Adamic A
AA = nx.adamic_adar_index(training_graph)
PA = nx.preferential_attachment(training_graph)

AA_as_dict= {frozenset((u,v)):aa for u,v,aa in AA}
```

In [175]:
```
#Let's select the pairs of nodes with highest values, print and plo
#We observe that they sounds like very likely edges to exist. (Addr
sorted_key_AA=sorted( AA_as_dict, key=AA_as_dict.get,reverse=True)[
sorted_key_AA
```

Out[175]:
```
[frozenset({'Bitstamp.net-old', 'ePay.info'}),
 frozenset({'Bitcoin.de', 'Bitstamp.net-old'}),
 frozenset({'BTC-e.com-old', 'Bitstamp.net-old'}),
 frozenset({'BTC-e.com-old', 'McxNOW.com'}),
 frozenset({'Bitfinex.com-old2', 'CoinTrader.net'}),
 frozenset({'BTC-e.com-old', 'BitcoinFog'}),
 frozenset({'Cex.io', 'CoinTrader.net'}),
 frozenset({'LocalBitcoins.com-ol', 'ePay.info'}),
 frozenset({'Bitfinex.com-old2', 'Dagensia.eu'}),
 frozenset({'BTC-e.com-old', 'Luno.com'})]
```

In [176]:
```
#Same analysis for preferntial attachment.
PA_as_dict= {frozenset((u,v)):cn for u,v,cn in PA}
sorted_key_PA=sorted( PA_as_dict, key=PA_as_dict.get,reverse=True)[
sorted_key_PA
```

Out[176]:
```
[frozenset({'Bitstamp.net-old', 'ePay.info'}),
 frozenset({'BTC-e.com-old', 'Bitstamp.net-old'}),
 frozenset({'CoinTrader.net', 'Cryptsy.com-old'}),
 frozenset({'Bitfinex.com-old2', 'CoinTrader.net'}),
 frozenset({'Bitstamp.net-old', 'Cryptsy.com-old'}),
 frozenset({'AnxPro.com', 'Cryptsy.com-old'}),
 frozenset({'LocalBitcoins.com-ol', 'ePay.info'}),
 frozenset({'BTC-e.com-old', 'LocalBitcoins.com-ol'}),
 frozenset({'Bitstamp.net-old', 'LocalBitcoins.com-ol'}),
 frozenset({'SilkRoad2Market', 'ePay.info'})]
```

In [ ]:
```
# We have to transform data in the right form
features_positive=[[PA_as_dict[np],AA_as_dict[np]] for np in traini
features_negatives=[[PA_as_dict[np],AA_as_dict[np]] for np in train
```

Logistic Classifier

• Value to predict : ‣ 0 (no edge) ‣ 1 (edge)

Minimize a cost function to find best parameters values.

Predict the most likely edges with the model.

In [177]:
```
from sklearn import linear_model,tree

#We can now create and train a Logistic classifier
model = linear_model.LogisticRegression()
predictor = model.fit(features_positive+features_negatives,[1]*len(

#Let's now predict most likely edges to appear. We use predict_prob
nodePairs=[frozenset((u,v)) for u in training_graph.nodes for v in
prediction = predictor.predict_proba([[PA_as_dict[np],AA_as_dict[np

#We sort the predictions from most likely to least likely, and plot
prediction=pd.DataFrame({"nodePair":nodePairs,"score":[x[0] for x i
prediction = prediction.sort_values("score",ascending=False)

prediction.head(20)
```

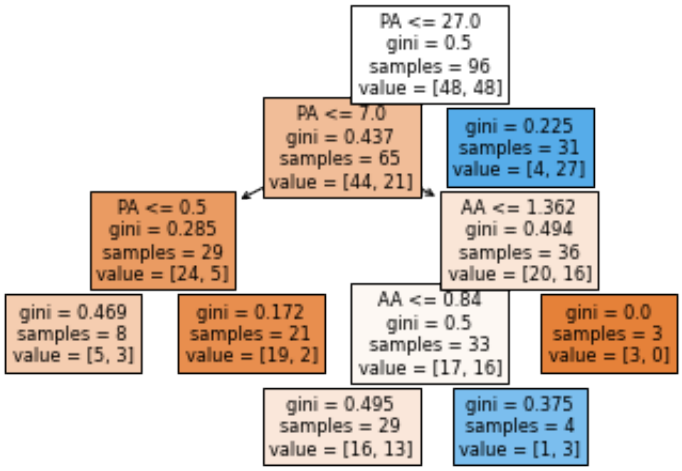
Out[177]:

	nodePair	score
1227	(Bitcoin.de-old, Justcoin.com)	0.770787
1475	(Justcoin.com, Bitcoin.de-old)	0.770787
1712	(158dPP7Z5Nb5J7CwBArx, BitBargain.co.uk)	0.770787
463	(BitBargain.co.uk, 158dPP7Z5Nb5J7CwBArx)	0.770787
89	(1KW4Wmo2XBw8X2sYtWEq, Poloniex.com)	0.767945
699	(1KW4Wmo2XBw8X2sYtWEq, Poloniex.com)	0.767945
1785	(158dPP7Z5Nb5J7CwBArx, 14o678qnBDKJuKnmD1bt)	0.760219
458	(BitBargain.co.uk, 1AtEhqyccigEgnzNWiAU)	0.760219
1743	(158dPP7Z5Nb5J7CwBArx, 14o678qnBDKJuKnmD1bt)	0.760219
1528	(158dPP7Z5Nb5J7CwBArx, 1AtEhqyccigEgnzNWiAU)	0.760219
1754	(BitBargain.co.uk, 14o678qnBDKJuKnmD1bt)	0.760219

Decision Tree Classifier

In [193]:
```
#Same thing, but using a Decision tree Classifier.
model = tree.DecisionTreeClassifier(max_leaf_nodes=6)
predictor = model.fit(features_positive+features_negatives,[1]*len(

tree.plot_tree(predictor, filled=True,feature_names=["PA","AA"])
```

Out[193]:
```
[Text(209.25, 195.696, 'PA <= 27.0\ngini = 0.5\nsamples = 96\nvalu
e = [48, 48]'),
 Text(167.4, 152.208, 'PA <= 7.0\ngini = 0.437\nsamples = 65\nvalu
e = [44, 21]'),
 Text(83.7, 108.72, 'PA <= 0.5\ngini = 0.285\nsamples = 29\nvalue
= [24, 5]'),
 Text(41.85, 65.232, 'gini = 0.469\nsamples = 8\nvalue = [5, 3]'),
 Text(125.55000000000001, 65.232, 'gini = 0.172\nsamples = 21\nval
ue = [19, 2]'),
 Text(251.10000000000002, 108.72, 'AA <= 1.362\ngini = 0.494\nsamp
les = 36\nvalue = [20, 16]'),
 Text(209.25, 65.232, 'AA <= 0.84\ngini = 0.5\nsamples = 33\nvalue
= [17, 16]'),
 Text(167.4, 21.744, 'gini = 0.495\nsamples = 29\nvalue = [16, 13]
'),
 Text(251.10000000000002, 21.744, 'gini = 0.375\nsamples = 4\nvalu
e = [1, 3]'),
 Text(292.95, 65.232, 'gini = 0.0\nsamples = 3\nvalue = [3, 0]'),
 Text(251.10000000000002, 152.208, 'gini = 0.225\nsamples = 31\nva
lue = [4, 27]')]
```



- Measure of heterogeneity (Gini, entropy…)
- Split recursively data in 2 to maximize homogeneity in child nodes

A classification tree learns a sequence of if then questions with each question involving one feature and one split point.

Plotting the tree. Can be read as follows (might be different if you re-run the code).

First, we observe at the top 48 positive examples and 48 negative examples. The first criterium is: if PA < 0.27, then predict no link. (44 negative examples, 21 positive ones).

It could potentially makes sense : high PA tends to be correlated with having edges

If PA is > 0.27, we have mostly positive examples, so we would tend to predict an edge.

If PA is > 0.7 but AA <= 1.362, then 20 of the training examples are negative, while only 16 are positive.

Although unintuitive, this can be understood as follows: if 2 Addresses are large (high PA) but have a moderate AA, then probably it means that they are not already connected. Thus it is unlikely that they will be connected in the future. We see the importance of Machine Learning here: it can discovert non-intuitive way to predict from data.

Graph Embedding

One of the limitations of graphs is the absence of vector features. However, we can learn an embedding of the graph.

There are several levels of embedding in a graph :

- Embedding graph components (nodes, edges, features…) (Node2Vec)
- Embedding sub-parts of a graph or a whole graph (Graph2Vec)

Node Embedding

We will follow "Node2Vec", a paper that was published by Aditya Grover and Jure Leskovec from Stanford University in 2016. A. Grover, J. Leskovec. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD), 2016.

According to the authors, "node2vec" is an algorithmic framework for representational learning on graphs. Given any graph, it can learn continuous feature representations for the nodes, which can then be used for various downstream machine learning tasks.

The model learns low-dimensional representations for nodes by optimizing a neighborhood preserving objective, using random walks.

In [180]:
```
import networkx as nx
from node2vec import Node2Vec

# Create a graph
graph = G

# Precompute probabilities and generate walks
node2vec = Node2Vec(graph, dimensions=64, walk_length=30, num_walks
# Embed nodes
model = node2vec.fit(window=10, min_count=1, batch_words=4)  # Any
```

```
HBox(children=(FloatProgress(value=0.0, description='Computing tra
nsition probabilities', max=45.0, style=Prog…
```

What can we do with this embedding ? One can for example identify the most similar node. It returns a list of the most similar nodes and the corresponding probabilities.

If the nodes have labels, we can train an algorithm based on the embedding and attach a label (node labeling, most similar node…).

In [186]:
```
# Look for most similar nodes of "Luno.com"
model.wv.most_similar('Bitstamp.net-old')  # Output node names are
```

Out[186]:
```
[('Cryptsy.com-old', 0.6288605332374573),
 ('Justcoin.com', 0.5948702096939087),
 ('Kraken.com', 0.5699283480644226),
 ('Bitfinex.com-old2', 0.5599614381790161),
 ('Cavirtex.com', 0.555366039276123),
 ('Cryptorush.in', 0.545860230922699),
 ('BTC-e.com-old', 0.5246950387954712),
 ('Bitcoin.de-old', 0.5203766226768494),
 ('CoinJar.com', 0.5110173225402832),
 ('Luno.com', 0.5092356204986572)]
```

Edge Embedding

Edges can also be embedded, and the embedding can be further used for classification.

In [187]:
```
# Embed edges using Hadamard method
from node2vec.edges import HadamardEmbedder

edges_embs = HadamardEmbedder(keyed_vectors=model.wv)

# Get all edges in a separate KeyedVectors instance - use with caut
edges_kv = edges_embs.as_keyed_vectors()
```

```
Generating edge features: 100%|██████████| 1035/1035.0 [00:00<00:00, 138410.43it/s]
```

Then, retrieve the vectors by specifying the name of the 2 linked nodes.

Again, we can retrieve the most similar edge, which can be used for missing edges prediction for example.

In [190]:
```
# Look for most similar edges - this time tuples must be sorted and
edges_kv.most_similar(str(('Bitstamp.net-old', 'Kraken.com')))
```

Out[190]:
```
[("('Bitstamp.net-old', 'Cex.io')", 0.8269777297973633),
 ("('Bitstamp.net-old', 'Cryptsy.com-old')", 0.7886377573013306),
 ("('Bitstamp.net-old', 'Huobi.com')", 0.77458655834198),
 ("('Cryptsy.com-old', 'Kraken.com')", 0.7680836915969849),
 ("('Bitstamp.net-old', 'Luno.com')", 0.7572905421257019),
 ("('Justcoin.com', 'Kraken.com')", 0.7509183287620544),
 ("('Cryptorush.in', 'Kraken.com')", 0.7472730278968811),
 ("('Bitstamp.net-old', 'Bitstamp.net-old')", 0.7430604696273804),
 ("('Kraken.com', 'Kraken.com')", 0.7368906736373901),
 ("('Bitstamp.net-old', 'Cryptorush.in')", 0.7260271310806274)]
```

Nodes Classification

Given a graph where some nodes are not labeled, we want to predict their labels. This is in some sense a semi-supervised learning problem.

One common way to deal with such problems is to make the assumption that there is a certain smoothness on the graph. The Smoothness assumption states that points connected via a path through high-density regions on the data are likely to have similar labels. This is the main hypothesis behind the Label Propagation Algorithm.

The Label Propagation Algorithm (LPA) is a fast algorithm for finding communities in a graph using network structure alone as its guide, without any predefined objective function or prior information about the communities.

Convolution Neural Networks (CNN)

These are deep neural networks used to analyze image data. They solve image processing tasks.

CNNs structures share weights, local connections and consist of many layer stacked together. These structural properties of a CNN are also shared within a GNN.

The shared weight property is important in graphs as it leads to a reduction in the computation cost. Many layers stacked together are able to capture meaningful features in graphs networks. The existence of local connections are what graphs networks are all about. They are locally connected structures.