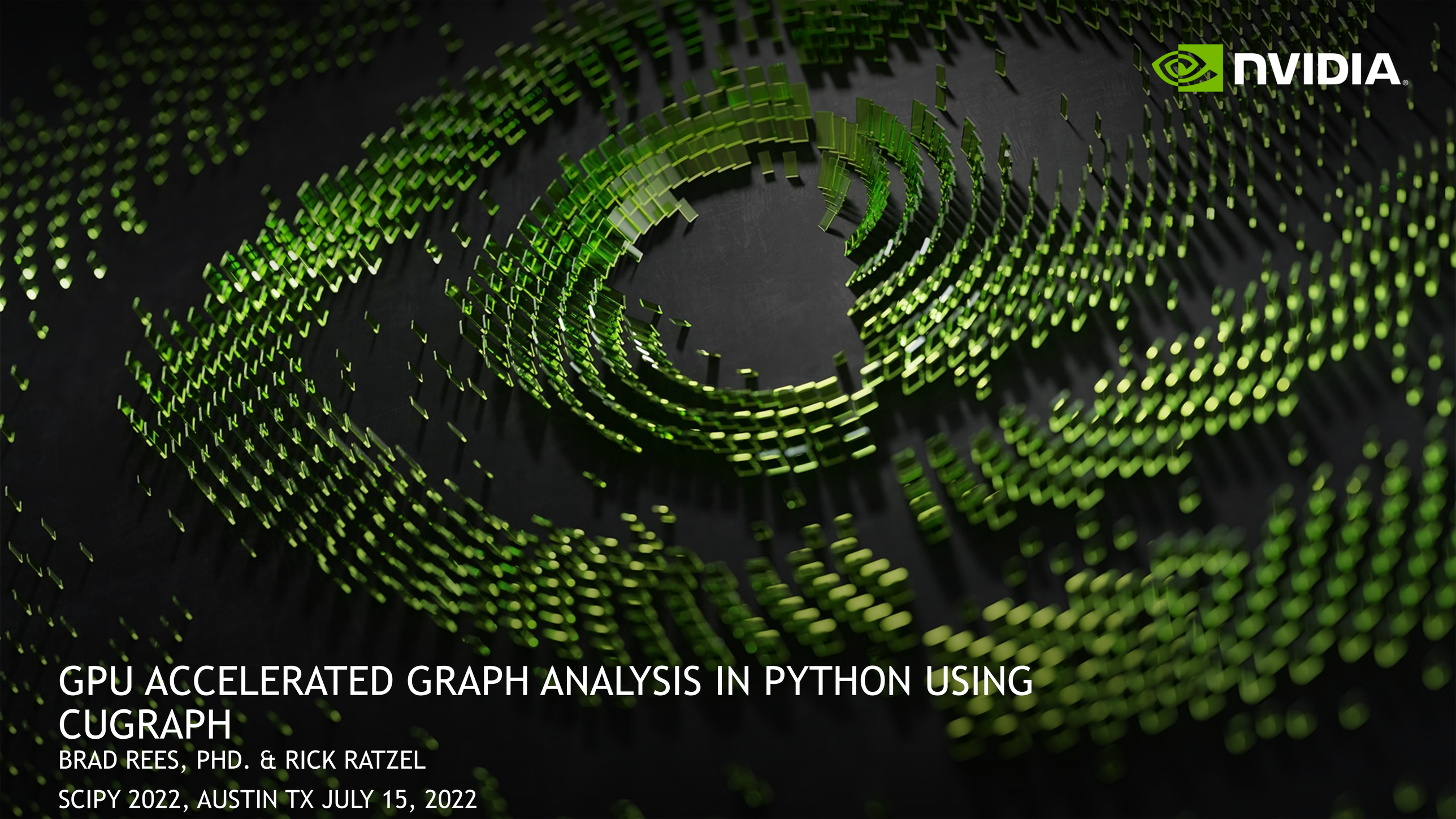


GPU ACCELERATED GRAPH ANALYSIS IN PYTHON USING CUGRAPH

BRAD REES, PHD. & RICK RATZEL

SCIPY 2022, AUSTIN TX JULY 15, 2022

AGENDA

- Introduction
  - Software Stack
  - Performance and Scaling
- Applying Graph Analysis

Brad Rees is a Senior Manager at NVIDIA and lead of the RAPIDS cuGraph team. Brad has been designing, implementing, and supporting a variety of advanced software and hardware systems for over 30 years. Brad specializes in complex analytic systems, primarily using graph analytic techniques for social and cyber network analysis. Brad has a Ph.D. in Computer Science from the Florida Institute of Technology.

Rick Ratzel is the Tech Lead for the Python / UX team. Rick joined NVIDIA just over three years ago, bringing several years of experience as a technical lead for teams in industries that include test and measurement, electronic design automation, and scientific computing. Rick's focus for cuGraph, and throughout his career, has been on software architecture and API usability.

Introduction

- RAPIDS - **rapids.ai and github.com/rapidsai**
  - suite of open-source software libraries that gives data scientists the freedom to execute end-to-end analytics workflows on GPUs
  - Launched in the fall of 2018

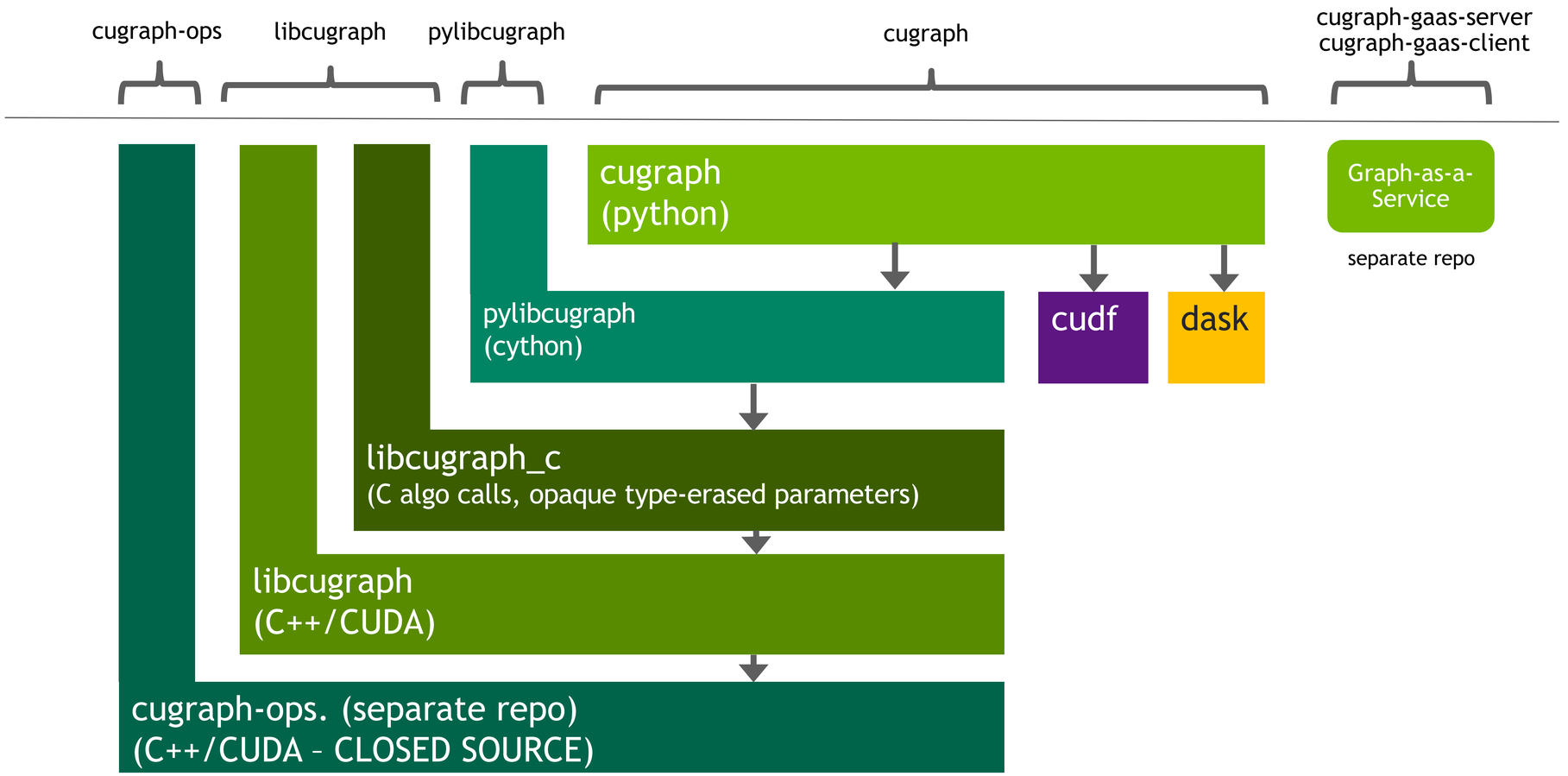
  - cuDF
    - Pandas-like
    - The ETL and data loading
    - utilizes a familiar DataFrame API.
  - cuML
    - Scikit-list
    - accelerated machine learning algorithms, like: XGBoost, RandomForest, UMAP, etc.
  - cuGraph
    - NetworkX-like
    - accelerated graph algorithms

Other libraries

- RAFT - Reusable Accelerated Functions and Tools
- cuSignal - signal processing
- cuspatial - GIS and spatiotemporal algorithms
- cuxfilter - accelerated cross filtering with cuDF
- and many more – see rapids.ai for list

THE CUGRAPH STACK

many layers of cugraph



Conda packages

Pip coming soon

At the top layer, we work to obfuscate how data is stored and processed.

A data scientist can just select an algorithms without concern for underlying structures.

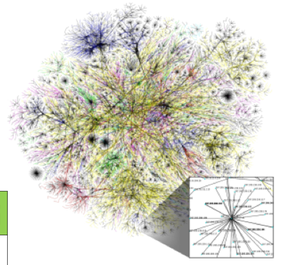
As you move down the stack, greater understanding of graphs and data structures is needed

cuGraph Vision Statement

Make graph analysis ubiquitous to the point that users just think in terms of analysis and not technologies or frameworks.

As of release 22.06

AVAILABLE ALGORITHMS



Class	Algorithms	MNMG
Centrality		
	Katz	Yes
	Betweenness Centrality	
	Edge Betweenness Centrality	
	Degree Centrality	Yes
	Eigenvector Centrality	Yes
Community		
	Leiden	
	Louvain	Yes
	Ensemble Clustering for Graphs	
	Spectral-Clustering - Balanced Cut	
	Spectral-Clustering - Modularity	
	Subgraph Extraction	
	Triangle Counting	Yes
	K-Truss	
Components		
	Weakly Connected Components	Yes
	Strongly Connected Components	
Core		
	K-Core	
	Core Number	
Layout		
	Force Atlas 2	
Linear Assignment		
	Hungarian	

Class	Algorithms	MNMG
Link Analysis		
	PageRank	Yes
	Personal PageRank	Yes
	HITS	Yes
Link Prediction		
	Jaccard Similarity	
	Weighted Jaccard Similarity	
	Overlap Similarity	
	Sorensen	
Traversal		
	Breadth First Search (BFS)	Yes
	Single Source Shortest Path (SSSP)	Yes
Sampling		
	Random Walks (Uniform and Biased)	
	EgoNet	
	Node2Vec	
	Neighborhood	Yes
Other		
	Minimum Spanning Tree	
	Maximum Spanning Tree	
Structure		
	Renumbering	Yes
	Symmetrize	Yes

MNMG == Multi-Node Multi-GPU

OVER 1 TRILLION EDGES !!!

cuGraph Scaling to Massive Scale

cuGraph scales smoothly from small graphs on 1 GPU to massive graphs with trillions of edges on 2,048 GPUs

- Being able to get answers in a matter of seconds regardless of graph scale if important
  - **PageRank:** Scale 36 (1.1 trillion directed edges) in 19.3 seconds (0.66 seconds per iteration, 2,048 GPUs)
  - **Louvain:** Scale 35 (0.55 trillion undirected edges or 1.1 trillion directed edges) in 336 seconds (1024 GPUs)
- Large scale testing just started
  - More analytics will be supported at 1000+ GPU scale
  - Continuous optimization for both memory usage and performance scalability

RMAT Data Generator using Graph500 Specs Edge Factor of 16

Scale	Number of Vertices	Number of Edges	COO Data Size (GB) in GPU
28	268,435,456	4,294,967,296	80
29	536,870,912	8,589,934,592	160
30	1,073,741,824	17,179,869,184	320
31	2,147,483,648	34,359,738,368	640
32	4,294,967,296	68,719,476,736	1,280
33	8,589,934,592	137,438,953,472	2,560
34	17,179,869,184	274,877,906,944	5,120
35	34,359,738,368	549,755,813,888	10,240
36	68,719,476,736	1,099,511,627,776	20,480
37	137,438,953,472	2,199,023,255,552	40,960

LET'S DIVE INTO USING CUGRAPH

Applying Graph Analysis

- The notebook can be found at:

  https://github.com/rapidsai-community/event-notebooks/tree/main/SCIPY_2022/cugraph_presentation

  just remember this part and you will be able to find notebooks

- The Notebook will NOT be run live
  - I never trust remote connections when doing a presentation
  - using PowerPoint since it is easy to annotate notebook

- Hardware:
  - HP Z840 Workstation
  - Dual Intel(R) Xeon(R) CPU E5-2643 v4 @ 3.40GHz
  - 64GB RAM
  - NVIDIA RTX A6000

- I will be showing some comparisons against NetworkX

if you are interested in SNA

https://www.insna.org/

THE DATA

Aguilar-Gallegos, Norman (2020), “Dataset on dynamics of Coronavirus on Twitter”, Mendeley Data, V1, doi: 10.17632/7ph4nx8hnc.1

data: https://data.mendeley.com/datasets/7ph4nx8hnc/1

I have no affiliation with the author and this dataset is just being used to illustrate graph analytics and not any derived insights from the data

[2]:
```
import cudf
import cugraph
```

Exploring the data

Tweet Types:

- 1.Tw Original Tweet
- 2.MT Mentioned within Tweet
- 3.RT Retweet
- 4.Re Replies

[3]:
```
# Let's load the "edges" dataset
gdf_tw_edges = cudf.read_csv("07a.Tw.edges.csv")
```

[4]:
```
# how much data
print("{:,}".format(len(gdf_tw_edges)))
```

```
7,296,841
```

[5]:
```
# Count the number of different Tweet types
gdf_tw_edges['Type'].value_counts()
```

[5]:
```
3. RT    5570466
4. Re    1025937
2. MT     700438
Name: Type, dtype: int32
```

[2]:
```
import cudf
import cugraph
```

Exploring the data

Tweet Types:

- 1.Tw Original Tweet
- 2.MT Mentioned within Tweet
- 3.RT Retweet
- 4.Re Replies

[2]:
```
%%timeit
gdf_tw_edges = cudf.read_csv("07a.Tw.edges.csv")
```

```
148 ms ± 1.57 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[3]:
```
%%timeit
pdf_tw_edges = pd.read_csv("07a.Tw.edges.csv")
```

```
6.84 s ± 31.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[3]:
```
# Let's load the "edges" dataset
gdf_tw_edges = cudf.read_csv("07a.Tw.edges.csv")
```

[4]:
```
# how much data
print("{:,}".format(len(gdf_tw_edges)))
```

```
7,296,841
```

[5]:
```
# Count the number of different Tweet types
gdf_tw_edges['Type'].value_counts()
```

[5]:
```
3. RT    5570466
4. Re    1025937
2. MT     700438
Name: Type, dtype: int32
```

Build a Graph

[5]:
```
# What does the data look like?
gdf_tw_edges.head()
```

[5]:

	from	to	Type	status_id	width
0	DrNancyM_CDC	CDCgov	2. MT	x1216747185565507585	1
1	WHO	WHOWPRO	4. Re	x1217027488137826304	1
2	CDCDirector	CDCgov	2. MT	x1217126105506820096	1
3	WHO	pr_moph	2. MT	x1217151178884222976	1
4	HelenBranswell	WHO	2. MT	x1217191264858206209	1

Since the from (source) and to (destination) columns are strings, we will need to use the renumbering feature in cuGraph to convert the strings to integer values. The nice part of the cuGraph renumbering feature is that it also converts the data back to the original string values in any output

[6]:
```
# Let's build an undirected graph
G = cugraph.Graph(directed=False)
G.from_cudf_edgelist(gdf_tw_edges, source='from', destination='to', renumber=True)
```

[7]:
```
(G.number_of_nodes(), G.number_of_edges())
```

[7]:
```
(2922882, 5894695)
```

Build a Graph

[5]:
```
# What does the data look like?
gdf_tw_edges.head()
```

[5]:

	from	to	Type	status_id	width
0	DrNancyM_CDC	CDCgov	2. MT	x1216747185565507585	1
1	WHO	WHOWPRO	4. Re	x1217027488137826304	1
2	CDCDirector	CDCgov	2. MT	x1217126105506820096	1
3	WHO	pr_moph	2. MT	x1217151178884222976	1
4	HelenBranswell	WHO	2. MT	x1217191264858206209	1

cuGraph allows the source (from) and destination (to) columns to be any data type – columns just need to be the same type

cuGraph does support multi-column as source/destination

directed=False – this means that cuGraph will symmetrize the data

Since the from (source) and to (destination) columns are strings, we will need to use the renumbering feature in cuGraph to convert the strings to integer values. The nice part of the cuGraph renumbering feature is that it also converts the data back to the original string values in any output

[6]:
```
# Let's build an undirected graph
G = cugraph.Graph(directed=False)
G.from_cudf_edgelist(gdf_tw_edges, source='from', destination='to', renumber=True)
```

[7]:
```
(G.number_of_nodes(), G.number_of_edges())
```

[7]:
```
(2922882, 5894695)
```

Build a Graph

[5]:
```
# What does the data look like?
gdf_tw_edges.head()
```

[5]:

	from	to	Type	status_id	width
0	DrNancyM_CDC	CDCgov	2. MT	x1216747185565507585	1
1	WHO	WHOWPRO	4. Re	x1217027488137826304	1
2	CDCDirector	CDCgov	2. MT	x1217126105506820096	1
3	WHO	pr_moph	2. MT	x1217151178884222976	1
4	HelenBranswell	WHO	2. MT	x1217191264858206209	1

If I was using the new Property Graph class, then I could save all the attributes. The basic "Graph" class only support numeric edge weights.
See Rick Ratzel's poster

Since the from (source) and to (destination) columns are strings, we will need to use the renumbering feature in cuGraph to convert the strings to integer values. The nice part of the cuGraph renumbering feature is that it also converts the data back to the original string values in any output

[6]:
```
# Let's build an undirected graph
G = cugraph.Graph(directed=False)
G.from_cudf_edgelist(gdf_tw_edges, source='from', destination='to', renumber=True)
```

[7]:
```
(G.number_of_nodes(), G.number_of_edges())
```

[7]:
```
(2922882, 5894695)
```

Build a Graph

[5]:
```
# What does the data look like?
gdf_tw_edges.head()
```

[5]:

	from	to	Type	status_id	width
0	DrNancyM_CDC	CDCgov	2. MT	x1216747185565507585	1
1	WHO	WHOWPRO	4. Re	x1217027488137826304	1
2	CDCDirector	CDCgov	2. MT	x1217126105506820096	1
3	WHO	pr_moph	2. MT	x1217151178884222976	1
4	HelenBranswell	WHO	2. MT	x1217191264858206209	1

Since the from (source) and to (destination) columns are strings, we will need to use the renumberi
integer values. The nice part of the cuGraph renumbering feature is that it also converts the data ba

[6]:
```
# Let's build an undirected graph
G = cugraph.Graph(directed=False)
G.from_cudf_edgelist(gdf_tw_edges, source='from', destination='to', renumber=True)
```

[7]:
```
(G.number_of_nodes(), G.number_of_edges())
```

[7]:
```
(2922882, 5894695)
```

The structure of the graph looks funny

$$avg\ degree = \frac{|Edges|}{|Nodes|} = \frac{5{,}894{,}695}{2{,}922{,}882} \cong 2$$

That indicates that there could be multiple disjoint subgraphs in the data

Let's look at the degree of the nodes. We can use:

- degree - total number of edges insident (connecected to) the vertex
- degrees - return both the in-degree and out-degree
- degree_centrality - same as 'degree'

Since this is an undirected graph, `degree` would count each edge twice (once for in and once for out)

[8]:
```
degree = G.degrees()
```

[9]:
```
degree.sort_values(by='out_degree', ascending=False).head(5)
```

[9]:

	in_degree	out_degree	vertex
2882791	56119	56119	OlaTinee
2672516	36534	36534	spectatorindex
13568	34876	34876	WHO
2829625	30196	30196	realDonaldTrump
25467	26421	26421	howroute

Wow, there are some very popular nodes!

[10]:
```
degree['out_degree'].describe().to_pandas().apply("{:,.2f}".format)
```

[10]:
```
count    2,922,882.00
mean             4.03
std             85.08
min              1.00
25%              1.00
50%              1.00
75%              2.00
max         56,119.00
Name: out_degree, dtype: object
```

75% of nodes have a degree of 2 or less!

Let's look at the degree of the nodes. We can use:

- degree - total number of edges insident (connecected to) the vertex
- degrees - return both the in-degree and out-degree
- degree_centrality - same as 'degree'

Since this is an undirected graph, `degree` would count each edge twice (once for in and once for out)

[8]:
```
degree = G.degrees()
```

[9]:
```
degree.sort_values(by='out_degree', ascending=False).head(5)
```

[9]:

	in_degree	out_degree	vertex
2882791	56119	56119	OlaTinee
2672516	36534	36534	spectatorindex
13568	34876	34876	WHO
2829625	30196	30196	realDonaldTrump
25467	26421	26421	howroute

Wow, there are some very popular nodes!

[10]:
```
degree['out_degree'].describe().to_pandas().apply("{:,.2f}".form
```

[10]:
```
count    2,922,882.00
mean             4.03
std             85.08
min              1.00
25%              1.00
50%              1.00
75%              2.00
max         56,119.00
Name: out_degree, dtype: object
```

75% of nodes have a degree of 2 or less!

Why not save the sort call to save time later?

```
degree = G.degrees()
degreeP = degree.to_pandas()
```

```
%%timeit
ds = degree.sort_values(by='out_degree', ascending=False)
```

```
6.57 ms ± 15.3 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
```

```
%%timeit
dps = degreeP.sort_values(by='out_degree', ascending=False)
```

```
205 ms ± 549 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Let's see how many subgraph (components) there could be. We use the Weakly Connected Component (WCC) algorithm

[11]:
```
comp = cugraph.weakly_connected_components(G)
```

[12]:
```
# Use groupby on the 'labels' column of the WCC output to get the counts of each connected component with the same label
label_count = comp.groupby('labels').count()
label_count.rename(columns={"vertex": "count"}, inplace=True)

print("Total number of components found : ", "{:,}".format(len(label_count)))
```

```
Total number of components found :  107,493
```

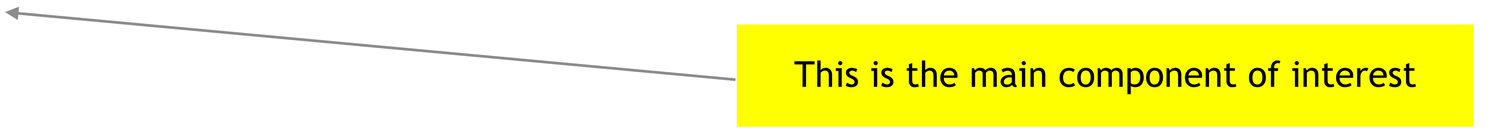
[13]:
```
# How many components have only two nodes?
len(label_count[label_count['count'] == 2])
```

[13]:
```
81883
```

[14]:
```
# what are the largest components?
label_count.sort_values(by='count', ascending=False).head()
```

[14]:

labels	count
2907067	2668807
2146571	83
2299021	69
2807631	51
2650010	49



Let's only look at the big component

In [16]:
```
# Get the label ID for the largest component
max_comp = label_count['count'].max()
big_comp_id = label_count[label_count['count'] == max_comp].index[0]
```

In [17]:
```
# now need to get the vertex IDs associated with those good components
big_comp = comp[comp['labels'] == big_comp_id]
```

In [18]:
```
# Now get the subgraph that just contains those nodes in large compoents
subgraph = cugraph.subgraph(G, big_comp['vertex'])
```

In [19]:
```
(subgraph.number_of_nodes(), subgraph.number_of_edges())
```

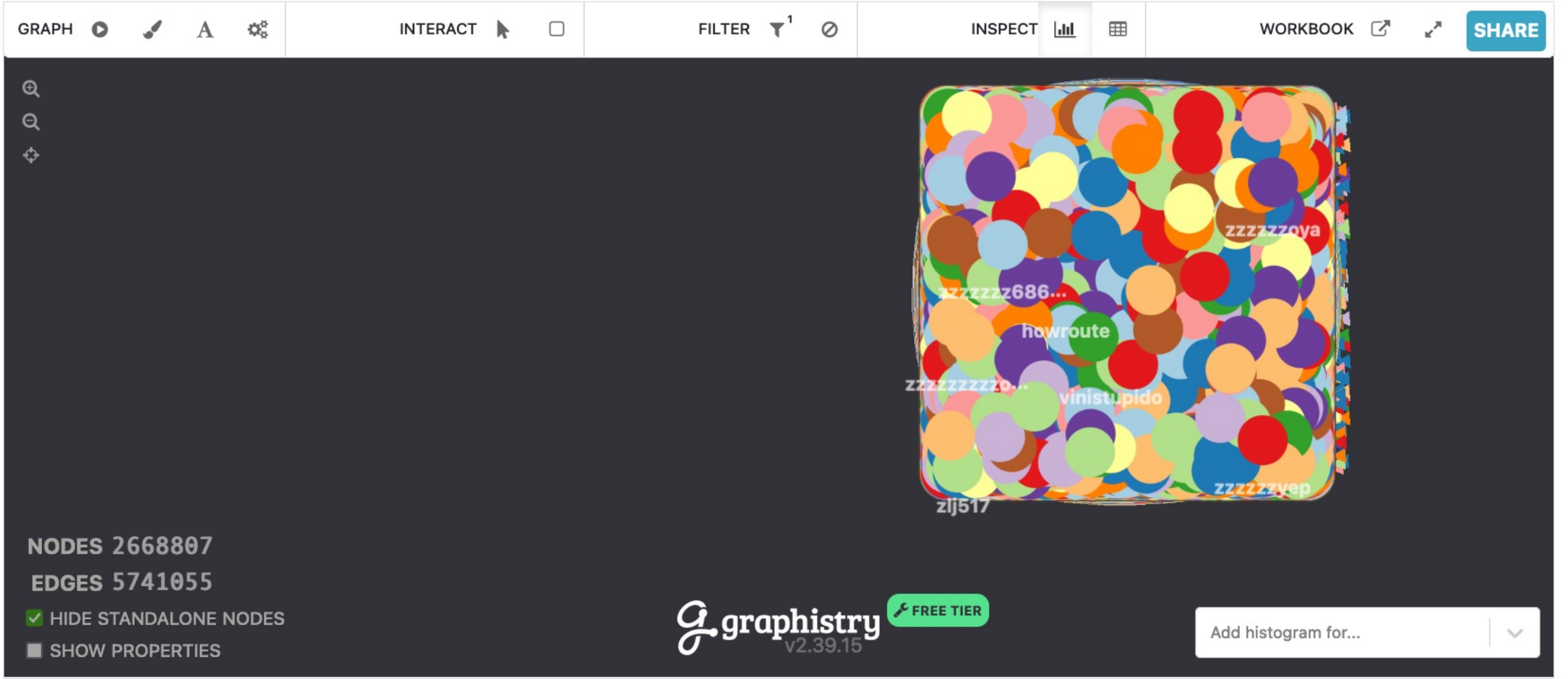
Out[19]:
```
(2668807, 5741055)
```

labels	count
1731182	2668807
2085265	83
1167083	69
2916221	51
2660936	49

```
import graphistry
```

```
graphistry.register(api=3, protocol="https", server="hub.graphistry.com", username=xxxxx, password=xxxxx)
```

```
graphistry.edges(subgraph.edges(), 'src', 'dst').plot()
```



Still way too much data to display

Who is important?

Centrality is the measure of how important, or cenbtral, a node is which in the graph **Note**: always set 'k' value Betweenness Centrality

[23]:
```
# Compute Centrality
# the centrality calls are very straight forward with the graph being the primary argument
# we are using the default argument values for all centrality functions
def compute_centrality(_graph) :
    # Compute Degree Centrality
    _d = cugraph.degree_centrality(_graph)

    # Compute the Betweenness Centrality
    _b = cugraph.betweenness_centrality(_graph, 100)

    # Compute Katz Centrality
    _k = cugraph.katz_centrality(_graph)

    # Compute PageRank Centrality
    _p = cugraph.pagerank(_graph,tol=0.0001)

    # Compute Eigenvector Centrality
    _e = cugraph.eigenvector_centrality(_graph)

    return (_d, _b, _k, _p, _e)
```

[24]:
```
# Print function
# The input is the tuple from the `compute_centrality` function
from IPython.display import display_html
def print_centrality(_C, _n):
    dc_top = _C[0].sort_values(by='degree_centrality', ascending=False).head(_n).to_pandas()
    bc_top = _C[1].sort_values(by='betweenness_centrality', ascending=False).head(_n).to_pandas()
    katz_top = _C[2].sort_values(by='katz_centrality', ascending=False).head(_n).to_pandas()
    pr_top = _C[3].sort_values(by='pagerank', ascending=False).head(_n).to_pandas()
    ec_top = _C[4].sort_values(by='eigenvector_centrality', ascending=False).head(_n).to_pandas()

    df1_styler = dc_top.style.set_table_attributes("style='display:inline'").set_caption('Degree').hide(axis='index')
    df2_styler = bc_top.style.set_table_attributes("style='display:inline'").set_caption('Betweenness').hide(axis='index')
    df3_styler = katz_top.style.set_table_attributes("style='display:inline'").set_caption('Katz').hide(axis='index')
    df4_styler = pr_top.style.set_table_attributes("style='display:inline'").set_caption('PageRank').hide(axis='index')
    df5_styler = ec_top.style.set_table_attributes("style='display:inline'").set_caption('Eigenvector').hide(axis='index')

    display_html(df1_styler._repr_html_()+
                 df2_styler._repr_html_()+
                 df3_styler._repr_html_()+
                 df4_styler._repr_html_()+
                 df5_styler._repr_html_(),
                 raw=True)
```

Who is important?

Centrality is the measure of how important, or cenbtral, a node is which in the graph **Note**: always set 'k' value Betweenness Centrality

[23]:
```
# Compute Centrality
# the centrality calls are very straight forward with the graph being the primary argument
# we are using the default argument values for all centrality functions
def compute_centrality(_graph) :
    # Compute Degree Centrality
    _d = cugraph.degree_centrality(_graph)

    # Compute the Betweenness Centrality
    _b = cugra
    # Compute
    _k = cugra
    # Compute
    _p = cugra
    # Compute
    _e = cugra
    return (_c
```

[25]:
```
C = compute_centrality(subgraph)
```

[26]:
```
print_centrality(C, 5)
```

Degree		Betweenness		Katz		PageRank		Eigenvector	
degree_centrality	**vertex**	**betweenness_centrality**	**vertex**	**katz_centrality**	**vertex**	**pagerank**	**vertex**	**eigenvector_centrality**	**vertex**
0.042056	OlaTinee	0.110956	WHO	0.001224	OlaTinee	0.006835	OlaTinee	0.269817	WHO
0.027379	spectatorindex	0.079125	generate_output	0.001011	spectatorindex	0.003161	spectatorindex	0.148587	business
0.026136	WHO	0.076862	OlaTinee	0.000993	WHO	0.002761	vinistupido	0.146949	nytimes
0.022629	realDonaldTrump	0.069667	spectatorindex	0.000942	realDonaldTrump	0.002161	YouTube	0.144140	howroute
0.019800	howroute	0.037128	CGTNOfficial	0.000900	howroute	0.001999	WHO	0.135622	CNN

The centrality numbers are very low, which could indicate that the graph needs to be further clustered

[24]:
```
# Print functi
# The input is
from IPython.d
def print_cent
    dc_top = _
    bc_top = _
    katz_top = _C[2].sort_values(by='katz_centrality', ascending=False).head(_n).to_pandas()
    pr_top = _C[3].sort_values(by='pagerank', ascending=False).head(_n).to_pandas()
    ec_top = _C[4].sort_values(by='eigenvector_centrality', ascending=False).head(_n).to_pandas()

    df1_styler = dc_top.style.set_table_attributes("style='display:inline'").set_caption('Degree').hide(axis='index')
    df2_styler = bc_top.style.set_table_attributes("style='display:inline'").set_caption('Betweenness').hide(axis='index')
    df3_styler = katz_top.style.set_table_attributes("style='display:inline'").set_caption('Katz').hide(axis='index')
    df4_styler = pr_top.style.set_table_attributes("style='display:inline'").set_caption('PageRank').hide(axis='index')
    df5_styler = ec_top.style.set_table_attributes("style='display:inline'").set_caption('Eigenvector').hide(axis='index')

    display_html(df1_styler._repr_html_()+
                 df2_styler._repr_html_()+
                 df3_styler._repr_html_()+
                 df4_styler._repr_html_()+
                 df5_styler._repr_html_(),
                 raw=True)
```

Community Detection

Component and Community are different but can sometimes be produce the same answer

Let's run Louvain

[27]:
```
communities_df, mod_score = cugraph.louvain(subgraph)
```

[28]:
```
#  Do we have a good clustering?  Look at the modularity score
mod_score
```

[28]:
```
0.7411910891532898
```

[29]:
```
# How man communities were found?
part_ids = communities_df["partition"].unique()
print("Louvain found " + str(len(part_ids)) + " communities")
```

```
Louvain found 1618 communities
```

[30]:
```
community_count = communities_df.groupby('partition').count()
community_count.rename(columns={"vertex": "count"}, inplace=True)
```

[31]:
```
# what are the largest communities?
community_count.sort_values(by='count', ascending=False).head()
```

[31]:

partition	count
1271	309076
1607	205208
503	171562
633	167905
1390	150457

Community Detection

Component != Community

Let's run Louvain

[27]:
```
communities_df, mod_score = cugraph.louvain(subgraph)
```

[28]:
```
#  Do we have a good clustering?  Look at the modularity score
mod_score
```

[28]:
```
0.7411910891532898
```

[29]:
```
# How man communities were found?
part_ids = communities_df["partition"].unique()
print("Louvain found " + str(len(part_ids)) + " communities")
```

```
Louvain found 1618 communities
```

[30]:
```
community_count = communities_df.groupby('partition').count()
community_count.rename(columns={"vertex": "count"}, inplace=True)
```

[31]:
```
# what are the largest communities?
community_count.sort_values(by='count', ascending=False).head()
```

[31]:

partition	count
1271	309076
1607	205208
503	171562
633	167905
1390	150457

```
%%timeit
_ = nx_comm.louvain_communities(Gnx)
```

```
7min 49s ± 33.9 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
%%timeit
communities, mod_score = cugraph.louvain(subgraph)
```

```
865 ms ± 1.9 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[32]:
```
def extract_subgraph(idx, sorted_cc, graph):
    _com_id = sorted_cc.index[idx]
    _v = communities_df[communities_df['partition'] == _com_id]
    _s = cugraph.subgraph(graph, _v['vertex'])
    return _s
```

[33]:
```
g0 = extract_subgraph(0, sorted_community_counts, subgraph)
print_centrality(compute_centrality(g0), 5)
```

Degree		Betweenness		Katz		PageRank		Eigenvector	
degree_centrality	**vertex**	**betweenness_centrality**	**vertex**	**katz_centrality**	**vertex**	**pagerank**	**vertex**	**eigenvector_centrality**	**vertex**
0.142458	realDonaldTrump	0.118139	realDonaldTrump	0.003597	realDonaldTrump	0.012928	realDonaldTrump	0.419529	realDonaldTrump
0.103723	howroute	0.105226	howroute	0.003108	howroute	0.009578	howroute	0.314915	howroute
0.078480	RealJamesWoods	0.070981	BNODesk	0.002789	RealJamesWoods	0.006905	RealJamesWoods	0.209167	IsChinar
0.062186	IsChinar	0.048290	IsChinar	0.002584	IsChinar	0.005073	Education4Libs	0.158311	BNODesk
0.061765	BNODesk	0.047223	DrEricDing	0.002579	BNODesk	0.005004	BNODesk	0.154515	jenniferatntd

[34]:
```
g1 = extract_subgraph(1, sorted_community_counts, subgraph)
print_centrality(compute_centrality(g1), 5)
```

Degree		Betweenness		Katz		PageRank		Eigenvector	
degree_centrality	**vertex**	**betweenness_centrality**	**vertex**	**katz_centrality**	**vertex**	**pagerank**	**vertex**	**eigenvector_centrality**	**vertex**
0.097687	CNNEE	0.126754	ActualidadRT	0.004415	CNNEE	0.011602	CNNEE	0.382461	CNNEE
0.092443	ActualidadRT	0.102663	CNNEE	0.004296	ActualidadRT	0.010421	ActualidadRT	0.336205	ActualidadRT
0.070495	AlertaNews24	0.084276	AlertaNews24	0.003800	AlertaNews24	0.008343	AlertaNews24	0.173120	AlertaNews24
0.056294	MaihenH	0.062752	dw_espanol	0.003479	MaihenH	0.005778	MaihenH	0.151726	dw_espanol
0.047796	dw_espanol	0.057830	ChalecosAmarill	0.003287	dw_espanol	0.004908	dw_espanol	0.136776	MaihenH

[35]:
```
g2 = extract_subgraph(2, sorted_community_counts, subgraph)
print_centrality(compute_centrality(g2), 5)
```

Degree		Betweenness		Katz		PageRank		Eigenvector	
degree_centrality	**vertex**	**betweenness_centrality**	**vertex**	**katz_centrality**	**vertex**	**pagerank**	**vertex**	**eigenvector_centrality**	**vertex**
0.213755	vinistupido	0.239208	vinistupido	0.004828	vinistupido	0.043346	vinistupido	0.659705	vinistupido
0.162788	AlineTosin	0.172298	AlineTosin	0.004252	AlineTosin	0.031361	AlineTosin	0.502406	AlineTosin
0.058335	Byano_DJ	0.124936	lucasrohan	0.003073	Byano_DJ	0.011662	Byano_DJ	0.180037	Byano_DJ
0.057332	celsolamounier	0.101950	jairbolsonaro	0.003061	celsolamounier	0.011165	celsolamounier	0.176943	celsolamounier
0.055432	lucasrohan	0.059221	Byano_DJ	0.003040	lucasrohan	0.009860	carloscrl144	0.171078	lucasrohan

We could have gotten to community from the start

[36]:
```
%%time
all_communities_df, mod_score = cugraph.louvain(subgraph)
```

```
CPU times: user 705 ms, sys: 159 ms, total: 864 ms
Wall time: 858 ms
```

[37]:
```
# How man communities were found?
all_part_ids = all_communities_df["partition"].unique()
print("Louvain found " + str(len(all_part_ids)) + " communities")
```

```
Louvain found 1618 communities
```

[38]:
```
all_community_count = all_communities_df.groupby('partition').count().rename(columns={"vertex": "count"})
```

[39]:
```
# what are the largest communities?
all_sorted_community_counts = all_community_count.sort_values(by='count', ascending=False)
all_sorted_community_counts.head(5)
```

[39]:

partition	count
1271	309076
1607	205208
503	171562
633	167905
1390	150457

[31]:

partition	count
1271	309076
1607	205208
503	171562
633	167905
1390	150457

We could have gotten to community from the start

[36]:
```
%%time
all_communities_df, mod_score = cugraph.louvain(subgraph)
```

```
CPU times: user 705 ms, sys: 159 ms, total: 864 ms
Wall time: 858 ms
```

[37]:
```
# How man communities were found?
all_part_ids = all_communities_df["partition"].unique()
print("Louvain found " + str(len(all_part_ids)) + " communitie
```

```
Louvain found 1618 communities
```

[38]:
```
all_community_count = all_communities_df.groupby('partition').
```

[39]:
```
# what are the largest communities?
all_sorted_community_counts = all_community_count.sort_values(by='count', ascending=False)
all_sorted_community_counts.head(5)
```

[39]:

partition	count
1271	309076
1607	205208
503	171562
633	167905
1390	150457

[ ]:
```
%%timeit
communities, mod_score = cugraph.louvain(G)
```

```
769 ms ± 2.39 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[ ]:
```
%%timeit
_ = nx_comm.louvain_communities(Gnx)
```

```
8min 37s ± 45.8 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

[31]:

partition	count
1271	309076
1607	205208
503	171562
633	167905
1390	150457

Looking for similarities

[40]:
```
Jaccard = cugraph.jaccard(g0)
```

[41]:
```
Jaccard.sort_values(by='jaccard_coeff', ascending=False).head()
```

[41]:

	jaccard_coeff	source	destination
37566	1.0	howroute	howroute
61005	1.0	IsChinar	IsChinar
65558	1.0	BNODesk	BNODesk
76222	1.0	jenniferatntd	jenniferatntd
96239	1.0	DarrenPlymouth	DarrenPlymouth

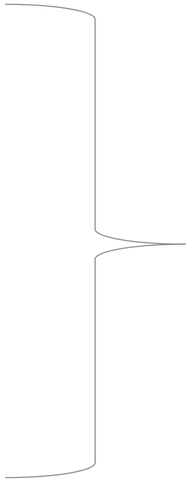
Lots of users referencing themselves

[42]:
```
# Drop self links  (should have done this at the graph level)
J2 = Jaccard[Jaccard['source'] != Jaccard['destination']]
```

[43]:
```
J2.sort_values(by='jaccard_coeff', ascending=False).head(10)
```

[43]:

	jaccard_coeff	source	destination
1009425	1.000000	GuyGuidoFawkes1	PutinRF_English
1011294	1.000000	PutinRF_English	GuyGuidoFawkes1
1183188	1.000000	800dbcloud	1NKDR0P
1185214	1.000000	1NKDR0P	800dbcloud
1298775	1.000000	cyberdefensemag	miliefsky
1316297	1.000000	miliefsky	cyberdefensemag
697283	0.918919	TresaBridges	lazyishhound
711856	0.918919	lazyishhound	TresaBridges
1181191	0.909091	REAL_DARGI	BLACKON13649707
1214883	0.909091	BLACKON13649707	REAL_DARGI



you also want to look at similarity of two-hop pairs for detecting multiple accounts

[44]:
```
Jaccard[Jaccard['source'] == 'realDonaldTrump'].sort_values(by='jaccard_coeff', ascending=False).head()
```

[44]:

	jaccard_coeff	source	destination
5643	0.051783	realDonaldTrump	PressSec
5635	0.042745	realDonaldTrump	POTUS
5636	0.040349	realDonaldTrump	BrianKolfage
5637	0.030850	realDonaldTrump	SecAzar
6065	0.024799	realDonaldTrump	BrianKarem

[45]:
```
Overlap = cugraph.overlap(g0)
```

[46]:
```
Overlap[Overlap['source'] == 'realDonaldTrump'].sort_values(by='overlap_coeff', ascending=False).head()
```

[46]:

	overlap_coeff	source	destination
7169	0.996350	realDonaldTrump	BrianKarem
6407	0.983333	realDonaldTrump	EbneHava
9498	0.980000	realDonaldTrump	marc_lotter
9583	0.958333	realDonaldTrump	BruhResign
27105	0.952381	realDonaldTrump	SteveScalise

Medium blog on Jaccard vs Overlap: https://medium.com/p/610e083b877d


cuGraph Repo: https://github.com/rapidsai/cugraph
Questions?
Thank You !!