Text analysis in 3D

ALEX RAZOUMOV
alex.razoumov@westdri.ca

Digital Research Alliance of Canada

SFU SIMON FRASER UNIVERSITY

Zoom controls

- Please mute your microphone and camera unless you have a question
- To ask questions at any time, type in Chat, or Unmute to ask via audio
  - please address chat questions to "Everyone" (not direct chat!)
- Raise your hand in Participants

- Email training@westdri.ca
- Our fall training schedule `https://bit.ly/wg2023b`
  - webinars, online courses, in-person workshops

- Some of today's examples come from my DHSI course "3D visualization for the humanities"
- Today's goal is not to present final workflows (they certainly are not!), but to show you some ideas and demos, and you could take it to the next level
  - how to go from texts to 3D graphs
  - how to use open-source scientific visualization packages for visualizing 3D graphs
  - how to scale this up to much larger networks
- In several places in the presentation I leave things as they are, without providing a solution, e.g.
  - finding common words between texts is slow in native Python – but we can find a solution if needed (there must be a Python library for that!)
  - there is no pretty+meaningful+fast layout in NetworkX to process networks with millions of nodes, but that does not mean you cannot write your own (and it's easy without a forced layout, e.g. you can put nodes on a grid based on their attributes)

From texts to 3D scatter plots

Semantic mapping

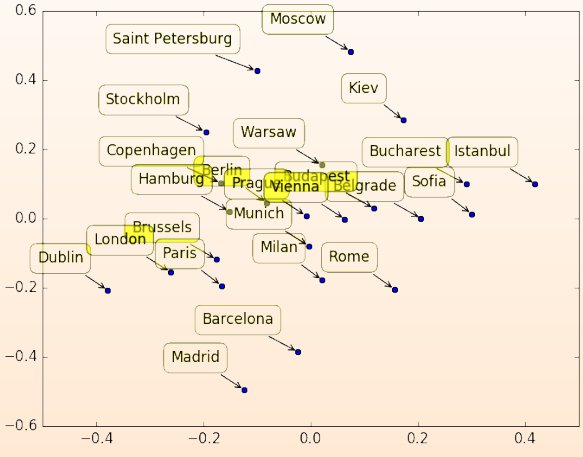
Idea inspired by this blog post from 2009

- Analyzed a corpus of 5,733,721 articles from 2,231 research journals (mostly science, technology and medical fields)
- Mapped the position of each journal in the 512-dimensional "semantic space" (more on this later)
- Calculated a 2231 × 2231 distance matrix in 512D
- Used multidimensional scaling to convert this matrix to 2D positions of 2231 points

- Coloured the points by 23 human-created journal categories
- Found excellent correspondence with human-created journal categories

Multidimensional scaling

Challenge: given a 24×24 table of pairwise distances between 24 cities, reconstruct their relative positions in 2D.



Semantic analysis of five public-domain texts

(1) The Time Machine, by Herbert Wells

(2) Oliver Twist, by Charles Dickens

(3) Adventures of Huckleberry Finn, by Mark Twain

(4) The War of the Worlds, by Herbert Wells

(5) Galilean-Invariant Cosmological Hydrodynamical Simulations on a Moving Mesh, by Volker Springel

(6) The Brothers Karamazov, by Fyodor Dostoevsky

- We'll analyze dictionaries and relative word frequencies and visualize a distance-based map of these texts in 3D

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

(5) Build a global dictionary (one for all five texts) of words, with N_{words} words

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

(5) Build a global dictionary (one for all five texts) of words, with N_{words} words

(6) Vectorize each paragraph in the N_{words}-dimensional space, positioning it according to its word count; for details see `http://radimrehurek.com/gensim/tut1.html`

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

(5) Build a global dictionary (one for all five texts) of words, with N_{words} words

(6) Vectorize each paragraph in the N_{words}-dimensional space, positioning it according to its word count; for details see `http://radimrehurek.com/gensim/tut1.html`

(7) Normalize each vector to the number of words in its paragraph, to count relative word frequencies

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

(5) Build a global dictionary (one for all five texts) of words, with N_{words} words

(6) Vectorize each paragraph in the N_{words}-dimensional space, positioning it according to its word count; for details see `http://radimrehurek.com/gensim/tut1.html`

(7) Normalize each vector to the number of words in its paragraph, to count relative word frequencies

(8) Calculate pairwise distances between all paragraphs in the N_{words}-dimensional space $\Rightarrow$ 180×180 matrix of numbers

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

(5) Build a global dictionary (one for all five texts) of words, with N_{words} words

(6) Vectorize each paragraph in the N_{words}-dimensional space, positioning it according to its word count; for details see `http://radimrehurek.com/gensim/tut1.html`

(7) Normalize each vector to the number of words in its paragraph, to count relative word frequencies

(8) Calculate pairwise distances between all paragraphs in the N_{words}-dimensional space $\Rightarrow$ 180×180 matrix of numbers

(9) Use multidimensional scaling to convert the distance matrix to paragraph positions in 3D, store them as VTK points

Algorithm

(1) From each text pick up 30 longest paragraphs, $\times 6$ texts $\Rightarrow$ 180 paragraphs

(2) Convert line breaks and dashes to spaces, remove punctuation

(3) Remove common words (prepositions, articles, etc)

(4) Count words across all paragraphs and remove words that appear only once across all texts

(5) Build a global dictionary (one for all five texts) of words, with N_{words} words

(6) Vectorize each paragraph in the N_{words}-dimensional space, positioning it according to its word count; for details see `http://radimrehurek.com/gensim/tut1.html`

(7) Normalize each vector to the number of words in its paragraph, to count relative word frequencies

(8) Calculate pairwise distances between all paragraphs in the N_{words}-dimensional space $\Rightarrow$ 180×180 matrix of numbers

(9) Use multidimensional scaling to convert the distance matrix to paragraph positions in 3D, store them as VTK points

(10) Visualize these points in 3D with ParaView, colouring by the author and sizing by the text per author (two texts for Herbert Wells)

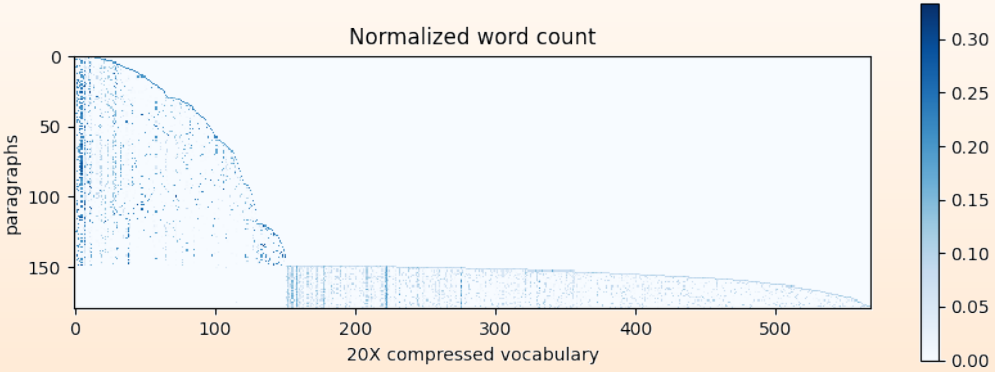
The code

1. The entire algorithm is implemented in `semanticMapping1.py` ☞ let's take a look at it
   - if working inside a Jupyter notebook, load the code into the current cell with `%load semanticMapping1.py` and then run it
   - if working in the terminal, use the command `python -i semanticMapping1.py`
2. Writing into a VTK file with

```
def writeNodesEdges(nodeCoords, scalar = [], name = [], power = [1,1], nodeLabel = [],
                    edges = [], method = 'vtkPolyData', fileout = 'test'):
"""
Store points and/or graphs as vtkPolyData or vtkUnstructuredGrid.
Required argument:
  - nodeCoords is an array of node coordinates (nnodes,3)
Optional arguments:
  - scalar is the list of attributes, each is the list of scalars for all nodes
  - name is the list of scalars' names
  - power is the scaling list for attributes: 1 for r~scalars, 0.333 for V~scalars
  - nodeLabel is a list of node labels
  - edges is a list of edges in the format [nodeID1,nodeID2]
  - method = 'vtkPolyData' or 'vtkUnstructuredGrid'
  - fileout is the output file name (will be given .vtp or .vtu extension)
"""
```

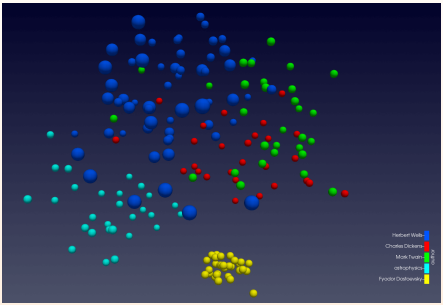
3. Open `texts.vtu` in ParaView

Plotting `normalizedFullCorpus`



Viewing results in ParaView

1. Colour glyphs by “author”
2. Switch from continuous to categorical colours and annotate them, e.g.
   - blue, author=1, Herbert Wells
   - red, author=2, Charles Dickens
   - green, author=3, Mark Twain
   - cyan, author=4, astrophysics
   - yellow, author=5, Fyodor Dostoevsky
3. Size glyphs by “novel per author” (small: The Time Machine, large: The War of the Worlds)



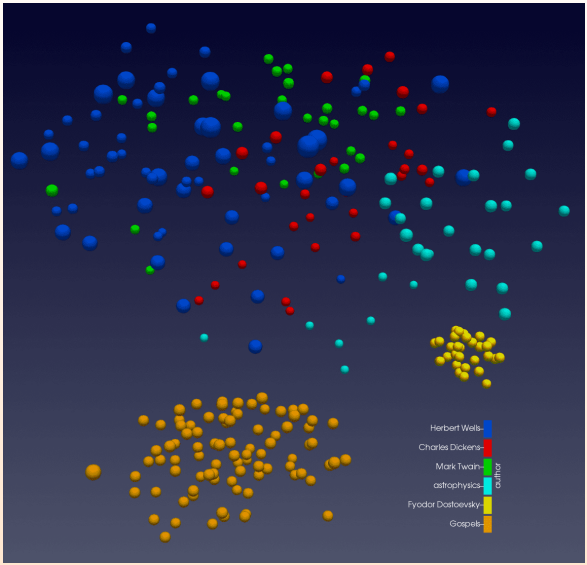
- Save the state to file `texts.pvsm`
- On Unix-like systems can reload from the GUI or from the command line with

```
/path/to/paraview --state=texts.pvsm
```

- Discuss reloading data
- Alternatively, we could map to 2D, using the third dimension to visualize some attribute, e.g. the publication year, or the text size, or the number of protagonists, etc. (will demo this later)

Let's add the four Gospels in Greek

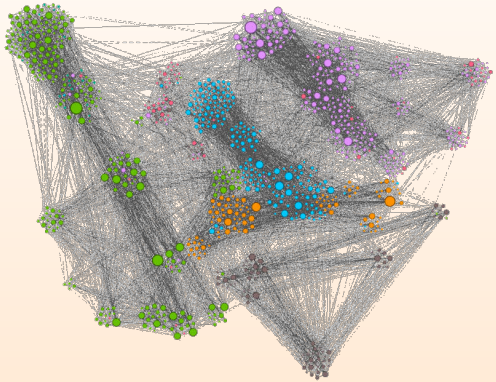
- Same workflow, now 10 texts (see `semanticMapping2.py`)
- Colour Gospels in orange



3D graphs

Dedicated 2D graph tools

- Many dedicated 2D tools, most popular ones are Gephi, Cytoscape (both open source)



- How can we extend this to 3D? And do we really want to?

Dedicated 3D graph tools

Looking for an interactive, open-source, cross-platform, currently maintained, user-friendly (and ideally in Python) dedicated 3D graph visualization tool

Dedicated 3D graph tools

Looking for an interactive, open-source, cross-platform, currently maintained, user-friendly (and ideally in Python) dedicated 3D graph visualization tool

- Surprisingly, there are very few ... most are either short-lived research projects and have not been updated in many years, or Windows only, or JavaScript-based, or commercial
- The most promising is `https://robert-haas.github.io/gravis-docs` (look for a 3D example here) but it has not been updated in ∼2 years ... thinking of doing a webinar on it next semester

Dedicated 3D graph tools

Looking for an interactive, open-source, cross-platform, currently maintained, user-friendly (and ideally in Python) dedicated 3D graph visualization tool

- Surprisingly, there are very few ... most are either short-lived research projects and have not been updated in many years, or Windows only, or JavaScript-based, or commercial
- The most promising is `https://robert-haas.github.io/gravis-docs` (look for a 3D example here) but it has not been updated in ∼2 years ... thinking of doing a webinar on it next semester
- Or we could combine existing general-purpose tools and packages: NetworkX + VTK + ParaView
  - advantages: (1) using general-purpose visualization tool; (2) everything is scriptable; (3) can scale directly to $10^{\sim 5.5}$ nodes, with a little extra care to $10^{\sim 9.5}$ nodes
  - disadvantages: graphs are static 3D objects, can't click on a node, highlight connections, move nodes, etc. (but we can script all these interactions!)
  - note: in the current implementation edges are displayed as straight lines; possible to use vtkArcSource or vtkPolyLine to create arcs and store them as vtkPolyData

(1) We'll use NetworkX + VTK to create a graph, position nodes, optionally compute graph statistics, and write everything to a VTK file; we'll do this in Python 3.11

(2) Load that file into ParaView

Dedicated 3D graph tools

Looking for an interactive, open-source, cross-platform, currently maintained, user-friendly (and ideally in Python) dedicated 3D graph visualization tool

- Surprisingly, there are very few ... most are either short-lived research projects and have not been updated in many years, or Windows only, or JavaScript-based, or commercial
- The most promising is `https://robert-haas.github.io/gravis-docs` (look for a 3D example here) but it has not been updated in ∼2 years ... thinking of doing a webinar on it next semester
- Or we could combine existing general-purpose tools and packages: NetworkX + VTK + ParaView
  - advantages: (1) using general-purpose visualization tool; (2) everything is scriptable; (3) can scale directly to $10^{\sim 5.5}$ nodes, with a little extra care to $10^{\sim 9.5}$ nodes
  - disadvantages: graphs are static 3D objects, can't click on a node, highlight connections, move nodes, etc. (but we can script all these interactions!)
  - note: in the current implementation edges are displayed as straight lines; possible to use vtkArcSource or vtkPolyLine to create arcs and store them as vtkPolyData

(1) We'll use NetworkX + VTK to create a graph, position nodes, optionally compute graph statistics, and write everything to a VTK file; we'll do this in Python 3.11

(2) Load that file into ParaView

- Alternatively, we could replace (NetworkX + VTK + ParaView) with (NetworkX + Plotly), but the result won't be as interactive / nice / scalable

NetworkX graphs

- NetworkX is a Python package for the creation, manipulation, and analysis of complex networks
- Documentation at `http://networkx.github.io`

```
import networkx as nx

# return all names (attributes and methods) inside nx
dir(nx)

# generate a list (of 139) built-in graph types
# with Python's ``list comprehension''
[x for x in dir(nx) if "_graph" in x]
```

NetworkX layouts ▷

```
# generate a (much shorter) list of built-in graph layouts
[x for x in dir(nx) if "_layout" in x]
# will print ['arf_layout', 'bipartite_layout', 'circular_layout',
#  'fruchterman_reingold_layout', 'kamada_kawai_layout', 'multipartite_layout',
#  'planar_layout', 'random_layout', 'rescale_layout', 'rescale_layout_dict',
#  'shell_layout', 'spectral_layout', 'spiral_layout', 'spring_layout']

# can always look at the help pages
help(nx.circular_layout)
```

- `spring_` and `fruchterman_reingold_` are the same, so really 13 built-in layouts
- Can use 3rd-party layouts, can create your own
- `circular_`, `random_`, `shell_` are fixed layouts
- `spring_` and `spectral_` are force-directed layouts: linked nodes attract each other, non-linked nodes are pushed apart

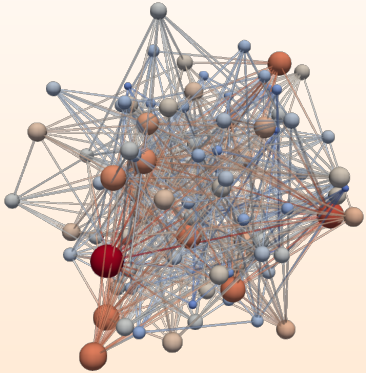
NetworkX layouts ▷▷

- Layouts typically return a dictionary, with each element being a 2D/3D coordinate array indexed by the node's number (or name)

```
# generate a random graph with 10 nodes and 50 edges
H = nx.gnm_random_graph(10,50)

# the layout is a dictionary of 2D coordinates of all 10 nodes
nx.shell_layout(H,dim=2)      # in this layout only dim=2 supported

# each value of these is an (x,y,z) coordinate of a node
nx.circular_layout(H,dim=3)
nx.spring_layout(H,dim=3)
nx.random_layout(H,dim=3)
nx.spectral_layout(H,dim=3)
```

Our first graph (`randomGraph1.py`)

```
import networkx as nx
from nodesAndEdges import writeNodesEdges

numberNodes, numberEdges = 100, 500
H = nx.gnm_random_graph(numberNodes,numberEdges)
print('nodes:', H.nodes())
print('edges:', H.edges())

# return a dictionary of positions keyed by node
pos = nx.random_layout(H,dim=3)

# convert to list of positions (each is a list)
xyz = [list(pos[i]) for i in pos]

degree = [d for i,d in H.degree()]
writeNodesEdges(xyz, edges=H.edges(), scalar=[degree],
                name=['degree'], fileout='network')
```



Labelling graph nodes

(1) Press V to bring up Find Data dialogue

(2) Data Producer = `network.vtp`, Element Type = Point

(3) Find Points with ID>=0 (or other selection), press Find Data

(4) Make `network.vtp` visible in the pipeline browser

(5) Check Point Labels -> ID

(6) Adjust the label font size

We can also label only few selected points, e.g. those with degree ≥ 10

Switch to spring layout

- Let's apply a force-directed layout

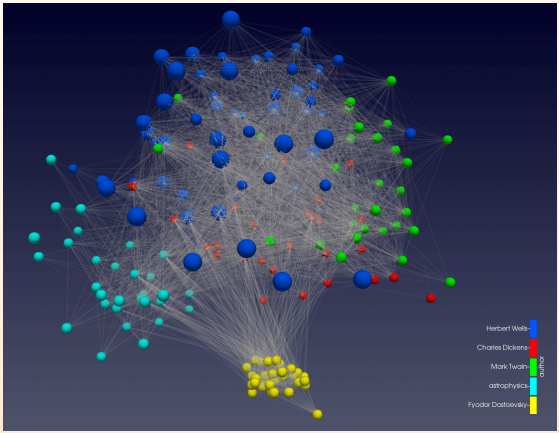
```
$ diff randomGraph{1,2}.py
10c10,11
< pos = nx.random_layout(H,dim=3)
---
> pos = nx.spring_layout(H,dim=3,k=1)
```

- Run "python randomGraph2.py" from the command line
- Reload the data

Adding graphs to our text visualizations

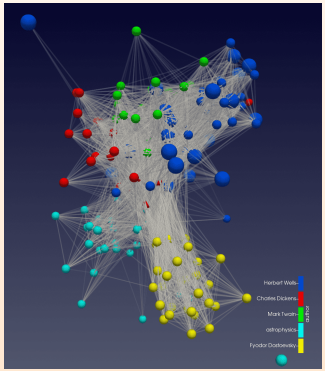
Connect close pairs

- In multidimensional scaling we already calculate pairwise distances
- Let's connect pairs with $d_{i,j} < 0.14$ (see `semanticMapping3.py`)

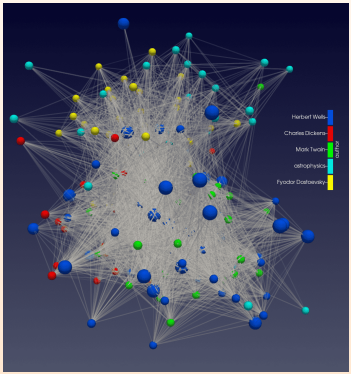


Apply a force layout

- Let's use only the connections (edges) to position the nodes: apply a force layout (see `semanticMapping4.py`)
- Pairwise distances $d_{i,j}$ now used only for edges, not for direct positioning
- Use the spring strength k to move the nodes closer or further apart



$k = 0.7$

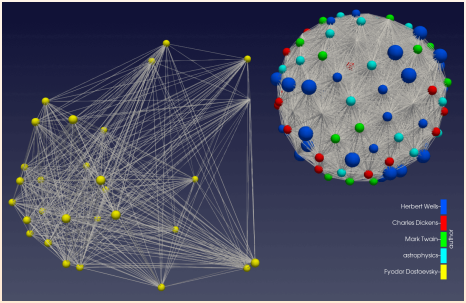


$k = 1.4$

Connect texts with common words

- Dostoevsky does not have any common words with the English texts!
- Pairwise distances $d_{i,j}$ use arbitrary scaling and are not a good measure ...
- Instead, let's connect pairs with at least 5 words in common
- $d_{i,j}$ will no longer be used $\Rightarrow$ can remove multidimensional scaling from the code
- See `semanticMapping5.py`

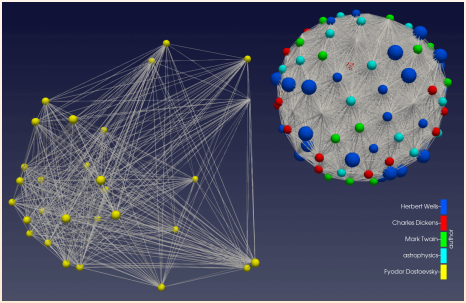
```
n, i = len(fullCorpus), -1
edges = []
for d1 in tqdm(fullCorpus):
    i += 1
    row = []
    for j, d2 in enumerate(fullCorpus):
        if i < j:
            if sum((d1!=0) * (d2!=0)) >=5:
                edges.append([i,j])
```



Connect texts with common words

- Dostoevsky does not have any common words with the English texts!
- Pairwise distances $d_{i,j}$ use arbitrary scaling and are not a good measure ...
- Instead, let's connect pairs with at least 5 words in common
- $d_{i,j}$ will no longer be used $\Rightarrow$ can remove multidimensional scaling from the code
- See `semanticMapping5.py`

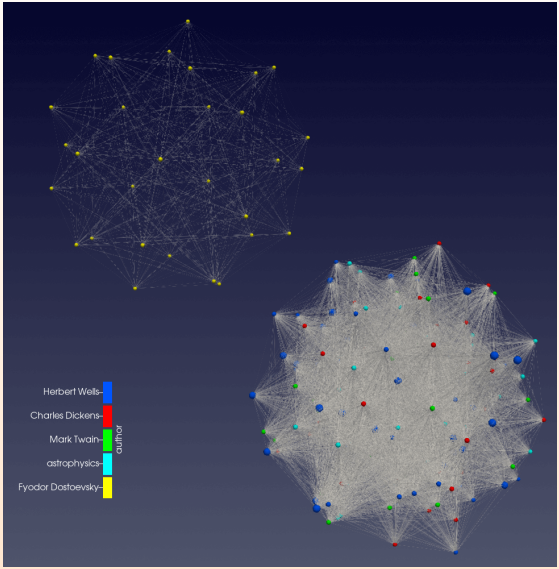
```
n, i = len(fullCorpus), -1
edges = []
for d1 in tqdm(fullCorpus):
    i += 1
    row = []
    for j, d2 in enumerate(fullCorpus):
        if i < j:
            if sum((d1!=0) * (d2!=0)) >=5:
                edges.append([i,j])
```



- Implemented in native Python $\Rightarrow$ quite slow compared to multidimensional scaling
  - for 600 paragraphs takes 6m vs. 5s for MD for the same texts

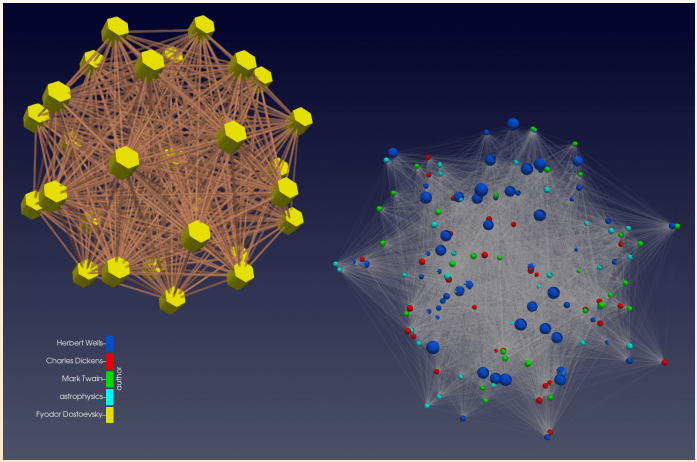
Separate layout for each language

- Makes no sense to use the same spring strength in both languages
- Apply a separate layout to Dostoevsky
- See `semanticMapping6.py`



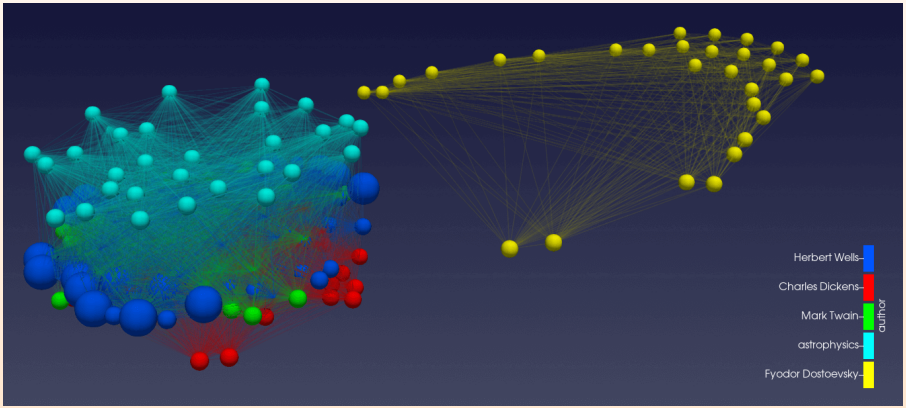
Separate visual properties for each language

- Two ways to assign different visuals to each graph:
  1. write into separate files (see `semanticMapping7.py`), or
  2. use ParaView filters



Encoding a variable in the 3rd dimension

- Spring layout to 2D
- Use the 3rd dimension to encode the year of the first publication (see `semanticMapping8.py`)



Scaling NetworkX + VTK + ParaView to bigger networks

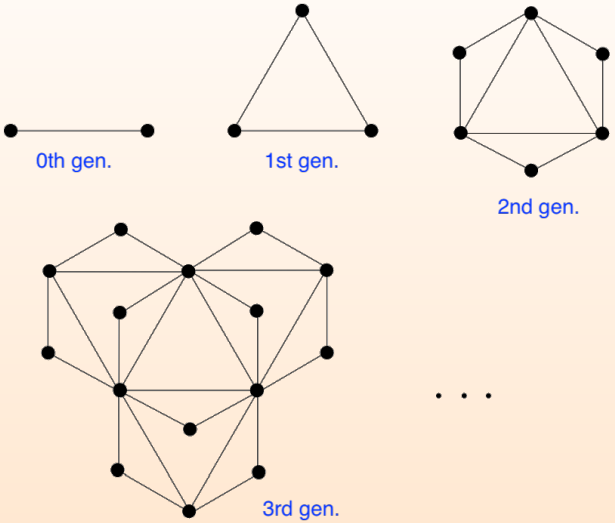
Scaling up

- This approach (NetworkX + VTK + ParaView) can scale without modification to $10^{\sim 5.5}$ nodes, with a little extra care to $10^{\sim 9.5}$ nodes
- On presenter's laptop see *mutOnCtOrbits.mp4* for a more complex graph ($600,000$ edges) created with this workflow
- Demo with a Dorogovtsev-Goltsev-Mendes graph

Dorogovtsev-Goltsev-Mendes graph

Dorogovtsev-Goltsev-Mendes graph is a fractal network from `http://arxiv.org/pdf/cond-mat/0112143.pdf`; in each subsequent generation:

1. every edge from the previous generation yields a new node, and
2. the new graph can be made by connecting together three previous-generation graphs

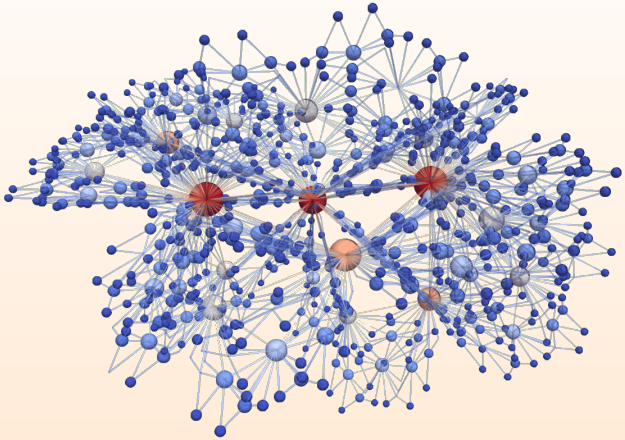


Dorogovtsev-Goltsev-Mendes graph (`dgm.py`)

- Generating the network: fast
- Computing the force layout: by far the most expensive part (for very large networks)
- Visualization: fast

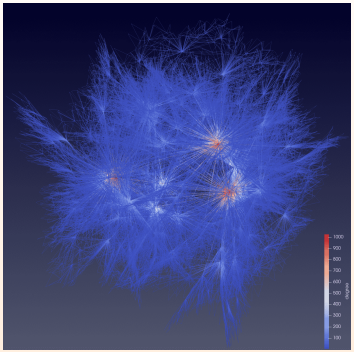
```
import networkx as nx, sys, locale
from nodesAndEdges import writeNodesEdges
generation = int(sys.argv[1])

H = nx.dorogovtsev_goltsev_mendes_graph(generation)

locale.setlocale(locale.LC_ALL, '')   # auto configuration
print(f'{nx.number_of_nodes(H):n}', 'nodes and', f'{nx.number_of_edges(H):n}', 'edges')

pos = nx.spring_layout(H, dim=3, k=0.003)   # slower
# pos = nx.spectral_layout(H, dim=3)        # faster, not as nice

xyz = [list(pos[i]) for i in pos] # list of positions (each is a list [x,y,z])

degree = [d for i,d in H.degree(H.nodes())]
writeNodesEdges(xyz, edges=H.edges(), scalar=[degree], name=['degree'], fileout='network9')
```

Dorogovtsev-Goltsev-Mendes graph (7th generation)

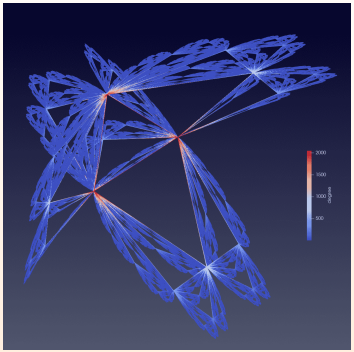


```
python dgm.py 7   # takes under 1s on a laptop
```

Dorogovtsev-Goltsev-Mendes graph: scaling up



- 10th gen
- stored in `network91.vtp`
- 29,526 nodes and 59,049 edges
- slower spring_layout (15m31s on a laptop)



- 11th gen
- stored in `network92.vtp`
- 88,575 nodes and 177,147 edges
- faster spectral_layout (2m6s on a laptop)

- Preparing for this presentation, I created a 15th-gen DGM graph: 7,174,455 nodes and 14,348,907 edges, takes few seconds to generate, easy to visualize in ParaView (e.g. with meaningless random_layout, few seconds to render a frame), but there is no pretty+meaningful+fast layout in NetworkX to process it with

Using NetworkX's built-in algorithms

Eigenvector centrality (`dgmCentrality.py`)

Let's compute and visualize eigenvector centrality in the 5th-deneration Dorogovtsev-Goltsev-Mendes graph with our custom 3D layout.

```
import networkx as nx
from nodesAndEdges import writeNodesEdges
H = nx.dorogovtsev_goltsev_mendes_graph(5)
pos = nx.spring_layout(H, dim=3)
print(nx.number_of_nodes(H), 'nodes and', nx.number_of_edges(H), 'edges')
degree = [d for i,d in H.degree(H.nodes())]
xyz = [[pos[i][0], pos[i][1], (degree[i])**0.5/5.7] for i in pos]

# compute and print eigenvector centrality
ec = nx.eigenvector_centrality(H) # dictionary of nodes with EC as the value
ecList = [ec[i] for i in ec]
print('degree =', degree)
print('eigenvector centrality =', ecList)
print('min/max =', min(ecList), max(ecList))

writeNodesEdges(xyz, edges=H.edges(), scalar=[degree,ecList], name=['degree', 'eigenvector centrality'],
                power=[0.333,0.333], fileout='network')
```

- Run `python dgmCentrality.py` and load into ParaView by hand
- Colour by degree, size by eigenvector centrality

Other statistics in NetworkX

- Various centrality measures: degree, closeness, betweenness, current-flow closeness, current-flow betweenness, eigenvector, communicability, load, dispersion
  `https://networkx.github.io/documentation/stable/reference/algorithms/centrality.html`

- Several hundred built-in algorithms for various calculations
  `https://networkx.github.io/documentation/stable/reference/algorithms`

Questions?