Graphs I — Breadth-First Search (BFS) ¶

Steven T. Smith ⟨stsmith@ll.mit.edu (mailto:Steven%20Thomas%20Smith%20%3Cstsmith@ll.mit.edu%3E)⟩

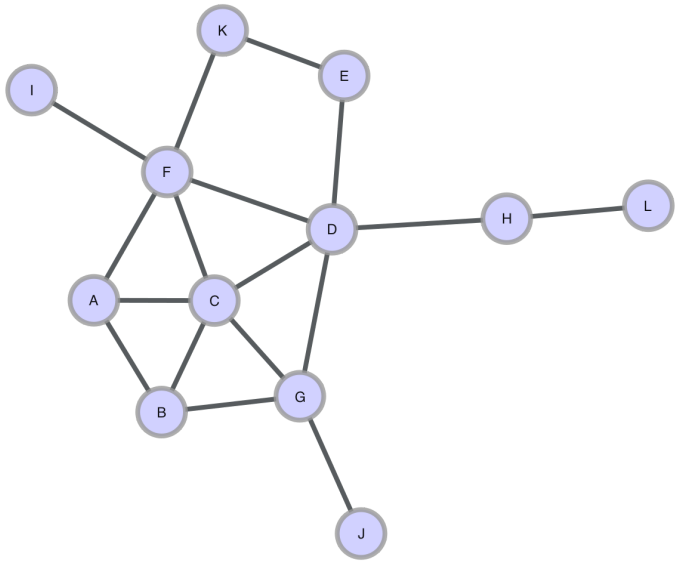
Lecture overview

- What's a graph?
- Why are graphs important and how are they used?
- How do we represent graphs?
- The breadth-first search (BFS) algorithm
- Extra: Markov chains and Perron-Frobenius

Graph background

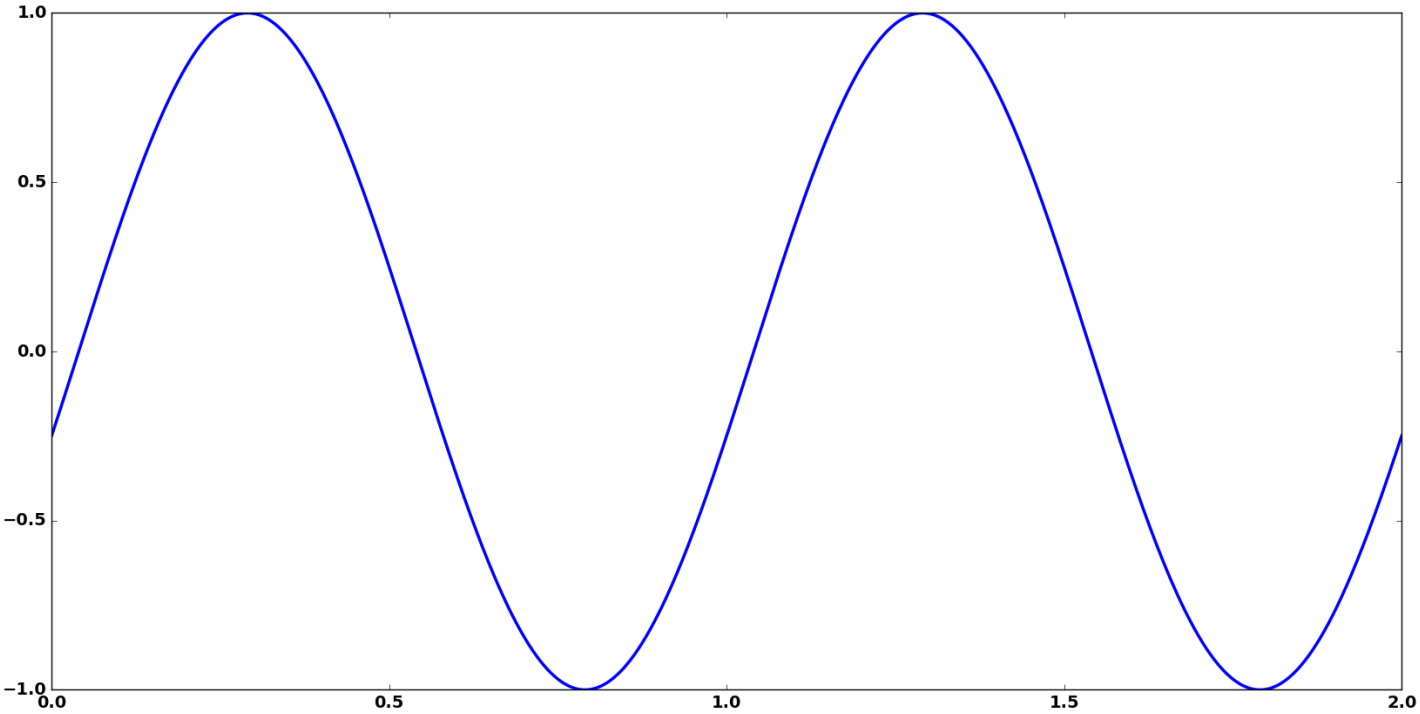
Definition. A graph is a set of relationships between a set of objects.

This is a random graph of some relationships between letters:



Steven T. Smith

We're not talking about this kind of graph!



Some mathematics

Simple graphs

Real definition. A ***simple*** graph

$$G = (V, E)$$

is defined by two sets, the set of N vertices

$$V = \{ v_1, v_2, \ldots, v_N \},$$

and the set of ***unordered*** K edges

$$E = \left\{ \{v_{e_{i1}}, v_{e_{o1}}\}, \{v_{e_{i2}}, v_{e_{o2}}\}, \ldots, \{v_{e_{iK}}, v_{e_{oK}}\} \right\} \subset \text{2-subsets of } V.$$

Steven T. Smith

How big is a graph?

- The ***order*** of the graph is $N = |V|$.
- The ***size*** of the graph is $K = |E|$.

Looking ahead, algorithms that cost around $O(N + K)$ are what we hope to achieve.

We'll abuse notation sometimes (oftentimes!) and refer to complexity in terms of $O(V)$ and $O(E)$—the number of vertices and edges.

$O(V^2)$ and $O(E^2)$ (or worse!) algorithms are really bad. If $|V|$ or $|E|$ is in the billions (common), then you need exaflops to do this. Perhaps the world's first exascale supercomputer will be the Tianhe-3 in 2020.

Better to look for $O(V)$ or $O(V \log V)$ methods!

We interrupt this lecture to insert some necessary Python premlimiaries (please ignore/skip over!).

In [1]:
```
import cairo, math, os, random, re, string
import numpy as np, numpy.random as npr, scipy.sparse as sparse

from IPython.core.display import HTML, display
from IPython.display import IFrame, YouTubeVideo
```

Steven T. Smith

In [2]:
```
# matplotlib and gt.graph_draw -- order important here
%matplotlib inline
import matplotlib as mpl
import matplotlib.pyplot as plt
import graph_tool.all as gt

# Legible plot style defaults
# http://matplotlib.org/api/matplotlib_configuration_api.html
# http://matplotlib.org/users/customizing.html
mpl.rcParams['figure.figsize'] = (10.0, 5.0)
mpl.rc('font',**{'family': 'sans-serif', 'weight': 'bold', 'size': 14})
mpl.rc('axes',**{'titlesize': 20, 'titleweight': 'bold', 'labelsize': 16,
'labelweight': 'bold'})
mpl.rc('legend',**{'fontsize': 14})
mpl.rc('figure',**{'titlesize': 16, 'titleweight': 'bold'})
mpl.rc('lines',**{'linewidth': 2.5, 'markersize': 18, 'markeredgewidth':
0})
mpl.rc('mathtext',**{'fontset': 'custom', 'rm': 'sans:bold', 'bf': 'sans:bo
ld', 'it': 'sans:italic', 'sf': 'sans:bold', 'default': 'it'})
# plt.rc('text',usetex=False) # [default] usetex should be False
mpl.rcParams['text.latex.preamble'] = [r'\usepackage{amsmath,sfmath} \boldm
ath']

figure_name = 'bfs_figure-'
figure_directory = 'images/'

# delete previous figure images
for f in os.listdir(figure_directory):
    if re.search(r'{}[0-9]+'.format(figure_name), f): os.remove(os.path.joi
n(figure_directory, f))

figure_number = 0
def get_figure_name(num=None):
    global figure_number
    if num is None: num = figure_number
    return os.path.join(figure_directory,figure_name + str(num) + '.png')
def increment_figure_number():
    global figure_number
    figure_number += 1
def display_figure_name(num=None):
    global figure_number
    if num is None:
        num = figure_number
        increment_figure_number()
    get_ipython().run_cell_magic(u'HTML', u'', u'<p style="text-align:cente
r;"><img src="{}"/></p>'.format(get_figure_name(num)))
```

In [3]:
```
for f in os.listdir(figure_directory):
    if re.search(r'{}[0-9]+'.format(figure_name), f): os.remove(os.path.joi
n(figure_directory, f))
```

Steven T. Smith

Back to the lecture!

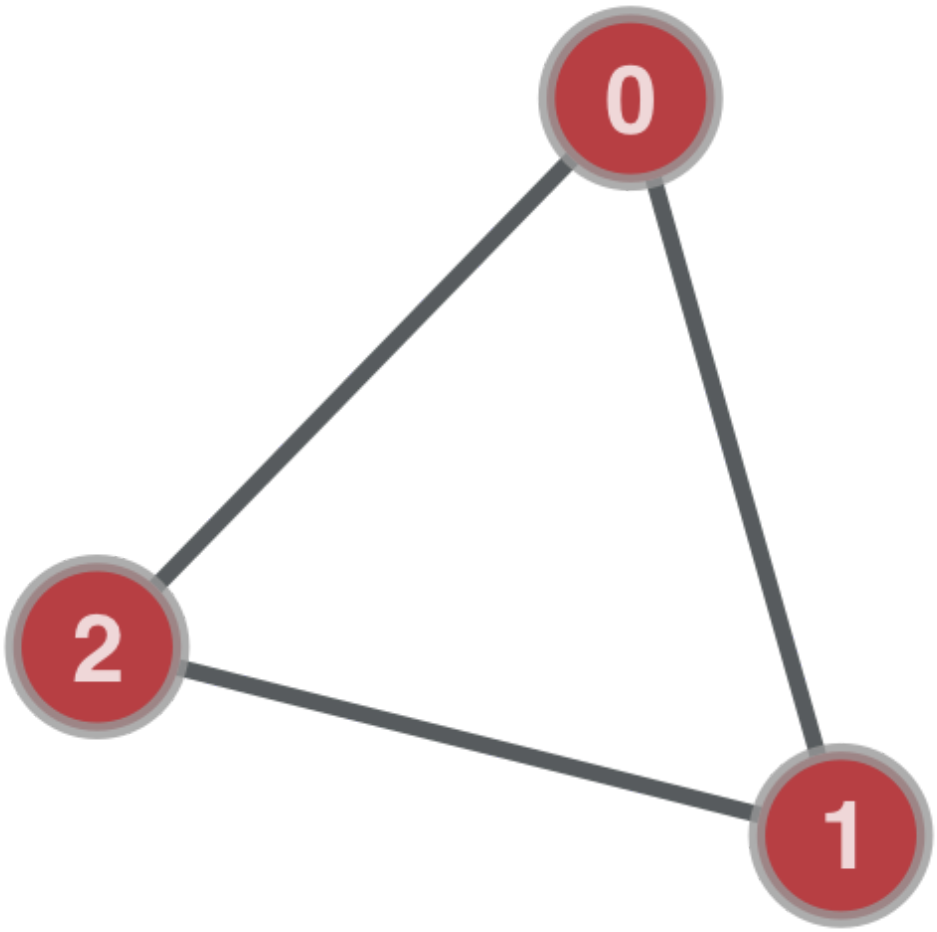
Here's an example simple graph:

In [4]:
```
edge_list = [ (0,1), (1,2), (2,0) ]

# graph-tool implementation
g = gt.Graph(directed=False)
g.add_edge_list(edge_list)
```

In [5]:
```
gd_font_size=48
gd_output_size=(512,512)

res = gt.graph_draw(g,
              vertex_text=g.vertex_index,
              vertex_font_size=gd_font_size,
              vertex_font_family='sans-serif',
              vertex_font_weight=cairo.FONT_WEIGHT_BOLD,
              output_size=gd_output_size,
              output=get_figure_name(),
              inline=False)
```

Steven T. Smith

In [6]:
```
display_figure_name()
```



/>

Directed graphs

Real definition. A ***directed*** graph

$$G = (V, E)$$

is defined by the set of vertices V and the set of ***ordered*** edges

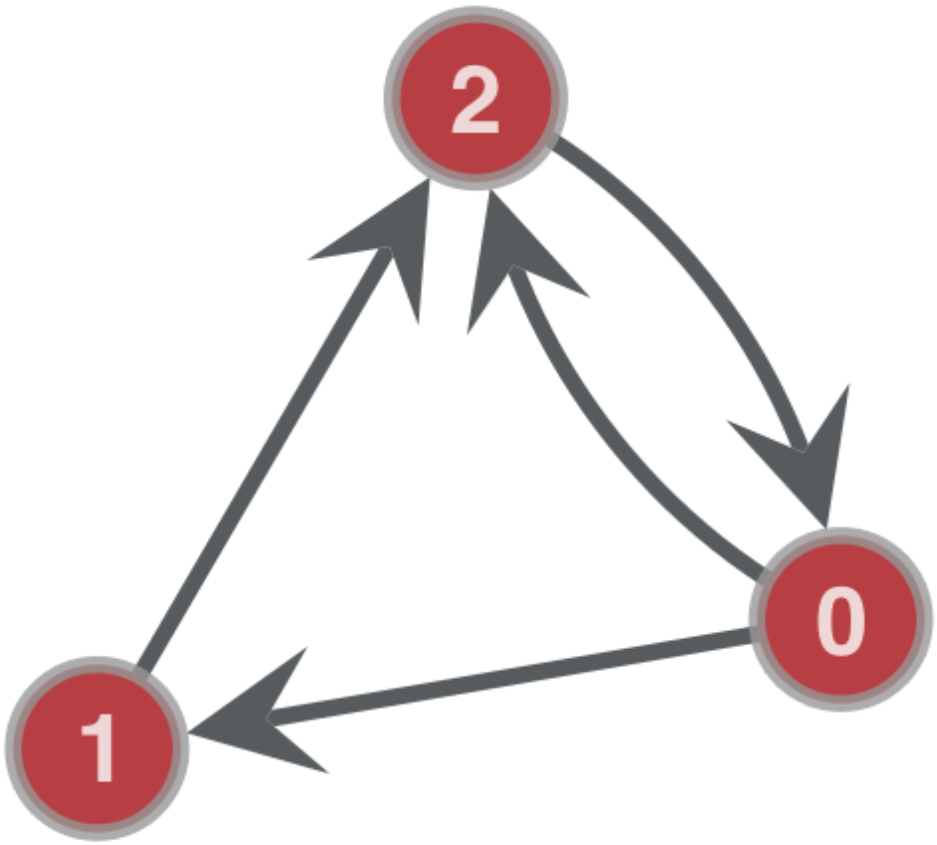
$$E = \left\{ (v_{e_{i1}}, v_{e_{o1}}), \ldots, (v_{e_{iK}}, v_{e_{oK}}) \right\} \subset V \times V.$$

Here's an example directed graph:

Steven T. Smith

In [7]:
```
g.set_directed(True)
g.add_edge_list([edge_list[-1][::-1]])  # add another edge
res = gt.graph_draw(g,
                    vertex_text=g.vertex_index,
                    vertex_font_size=gd_font_size,
                    vertex_font_family='sans-serif',
                    vertex_font_weight=cairo.FONT_WEIGHT_BOLD,
                    output_size=gd_output_size,
                    output=get_figure_name(),
                    inline=False)
```

In [8]:
```
display_figure_name()
```



/>

Steven T. Smith

Special kinds of graphs

- A ***subgraph*** $G' \subseteq G$ is a graph (V', E') with $V' \subseteq V$ and $E' \subseteq E$.
- A ***connected*** graph is a graph for which a path exists between any two vertices.
- A ***planar*** graph is a graph that can by drawn on a plane without crossing edges.
- An ***acyclic*** graph is a graph that doesn't contain any closed paths $v_0 \to v_{p_1} \to \cdots \to v_{p_{L-1}} \to v_0$.
  - A ***directed acyclic graph*** **(DAG)** is an important case

- A ***clique*** or ***complete*** graph K is a simple graph in which every pair of vertices is connected by an edge. N.b. $|E_{K_N}| = N(N-1)/2$.
- A ***bipartite*** graph is a graph such that $V = A \coprod B$ (disjoint union) and $E \subset A \times B$, i.e. every edge has its ends in different sets.
- A ***line graph*** of a graph G is a graph $L(G)$ in which the vertices are the edges of G and two vertices are adjacent if they have a common vertex in G.
- ***...*** (lots and lots of differents kinds of graphs …)

Steven T. Smith

Some important graph terminology

- The ***degree*** of a vertex is the number of its neighbors
  - In degree 👈 Out degree 👉 Total degree
- The ***degree vector*** $\mathbf{d}$ is the (column) vector of vertex degrees,
$$\mathbf{d} = (d_{v_1}, d_{v_2}, \ldots, d_{v_N})^{\mathrm{T}}$$
- The ***adjacency matrix*** is the V-by-V, $(0,1)$-matrix $\mathbf{A}$ with $A_{ij} = 1$ iff $(v_i, v_j) \in E$.
- The ***incidence matrix*** is the V-by-E, $(0,\pm 1)$-matrix $\mathbf{B}$ with $B_{ve} = \pm 1$ iff v is terminal/initial vertex of e.

Examples of adjacency and incidence matrices for simple and undirected graphs:

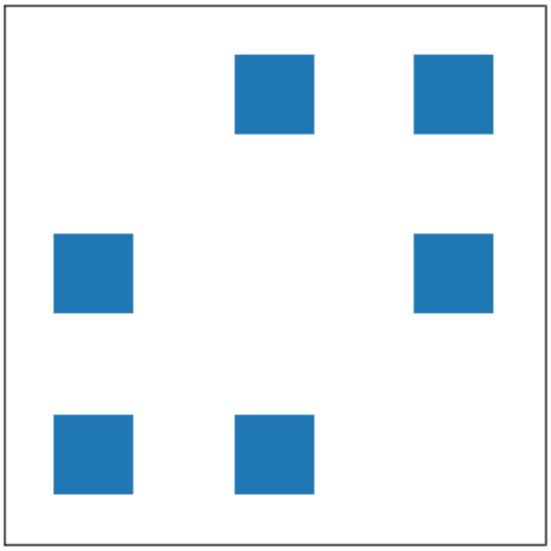
In [9]:
```
print('Edges of g:')
for e in g.edges(): print(e)
```

```
Edges of g:
(0, 1)
(0, 2)
(1, 2)
(2, 0)
```

Steven T. Smith

In [10]:
```
print('Adjacency matrix of a simple graph (symmetric!)')
g.set_directed(False)
print(gt.adjacency(g).todense())
print('(Also note the weighting for the additional edge.)')
plt.spy(gt.adjacency(g),markersize=40)
plt.gca().get_xaxis().set_visible(False);
plt.gca().get_yaxis().set_visible(False)
```

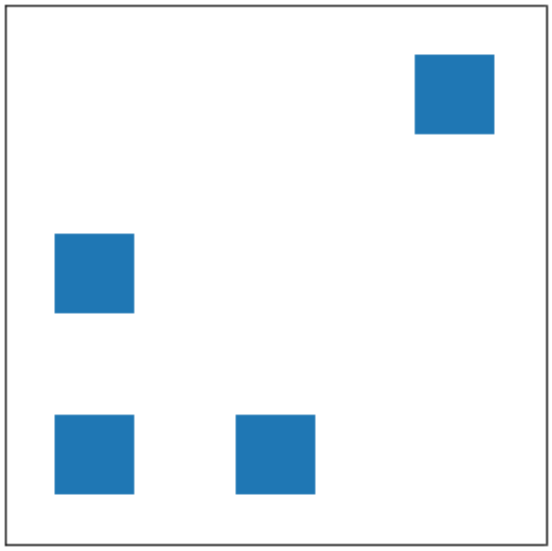
```
Adjacency matrix of a simple graph (symmetric!)
[[ 0.  1.  2.]
 [ 1.  0.  1.]
 [ 2.  1.  0.]]
(Also note the weighting for the additional edge.)
```



Steven T. Smith

In [11]:
```
print('Adjacency matrix of a directed graph (not symmetric!)')
g.set_directed(True)
print(gt.adjacency(g).todense())
plt.spy(gt.adjacency(g),markersize=40)
plt.gca().get_xaxis().set_visible(False);
plt.gca().get_yaxis().set_visible(False)
plt.savefig(get_figure_name())
```

```
Adjacency matrix of a directed graph (not symmetric!)
[[ 0.  0.  1.]
 [ 1.  0.  0.]
 [ 1.  1.  0.]]
```



Steven T. Smith

Random graphs

- A ***random graph*** is a graph whose edges are random.
- An ***Erdős–Rényi*** graph $G(n,p)$ is a random graph of order n whose edges are drawn from indepdendent Bernoulli trials with probability p.
  - For each of the possible $\binom{n}{2}$ edges, flip a p-coin.
  - $p=0$: The "null" graph; $p=1$: The complete graph

Cool fact*: $G(n,p)$ is almost surely connected if $p > n^{-1}\log n$ as $n \to \infty$, and is almost surely disconnected if $p < n^{-1}\log n$.

*P. Erdös and A. Rényi, “On the evolution of random graphs,” *Pubs. Math. Inst. Hung. Acad. Sci.*, vol. 5, pp. 17–61 (1960).

Random Erdős–Rényi Graph Examples

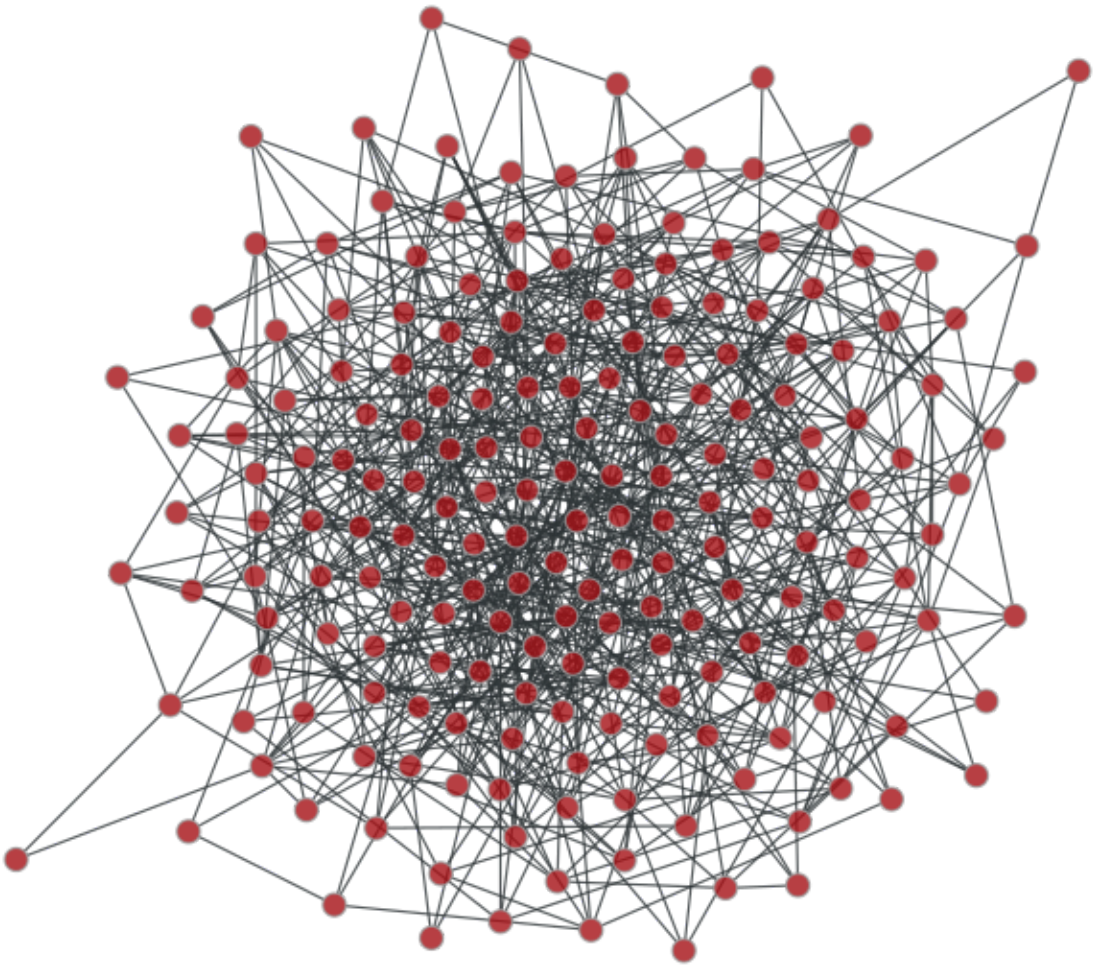
In [12]:
```
N = 200
p = 1.5 * np.log(N)/N

print('An almost surely connected graph (p = {}*log(n)/n):'.format(p*N/np.log(N)))
g = gt.random_graph(N, deg_sampler=lambda: npr.poisson((N-1)*p), directed=False, model='erdos')
res = gt.graph_draw(g,output=get_figure_name(),inline=False)
```

An almost surely connected graph (p = 1.5*log(n)/n):

An almost surely connected Erdős–Rényi graph

Steven T. Smith

In [13]:
```
display_figure_name()
```



/>

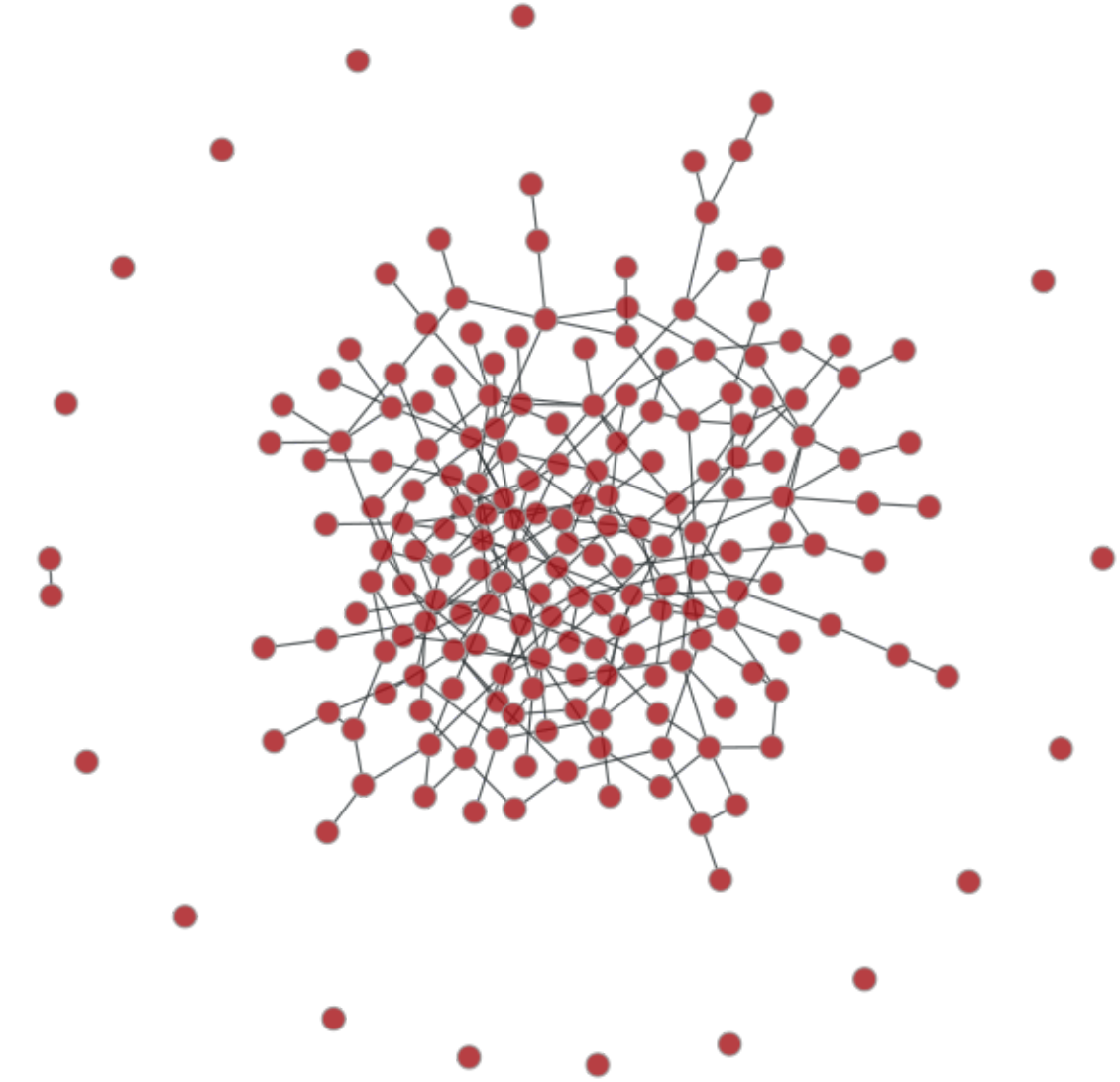
In [14]:
```
p = 0.5 * np.log(N)/N

print('An almost surely unconnected graph (p =
{}*log(n)/n):'.format(p*N/np.log(N)))
g = gt.random_graph(N, deg_sampler=lambda: npr.poisson((N-1)*p), directed=F
alse, model='erdos')
res = gt.graph_draw(g,output=get_figure_name(),inline=False)
```

An almost surely unconnected graph (p = 0.5*log(n)/n):

An almost surely disconnected Erdős–Rényi graph

Steven T. Smith

In [15]:
```
display_figure_name()
```

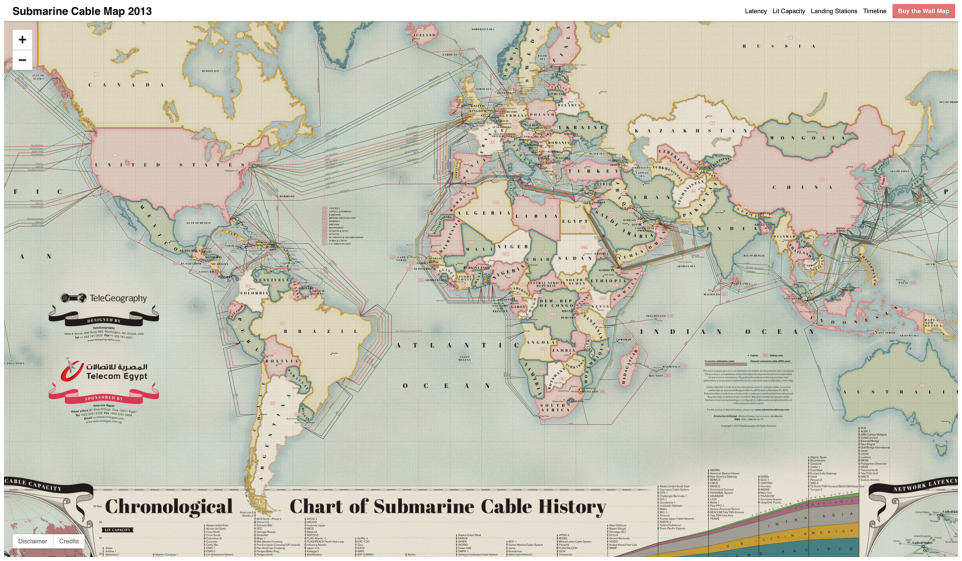


/>

Steven T. Smith

Graph applications

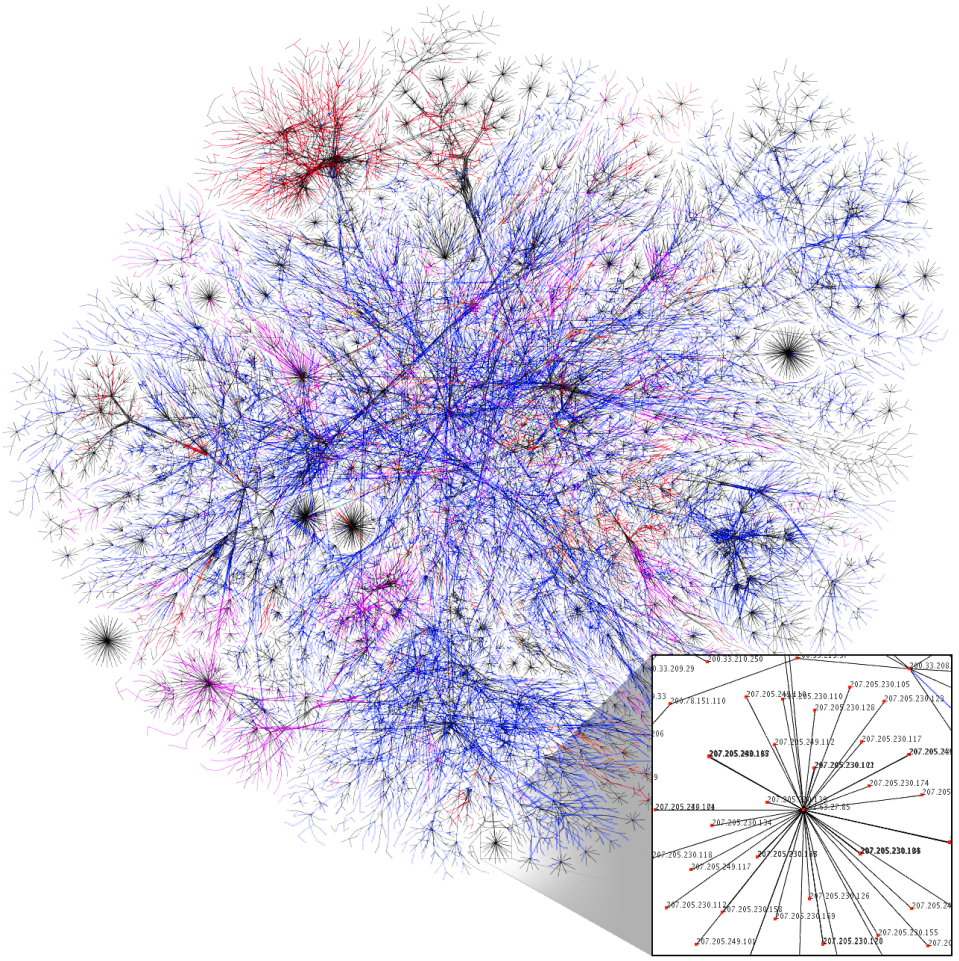
- Communications networks
  - Submarine cable map
  - Size 10^2
  - > 50 Tbps



Submarine Cable Map 2013 (http://submarine-cable-map-2013.telegeography.com)

Steven T. Smith

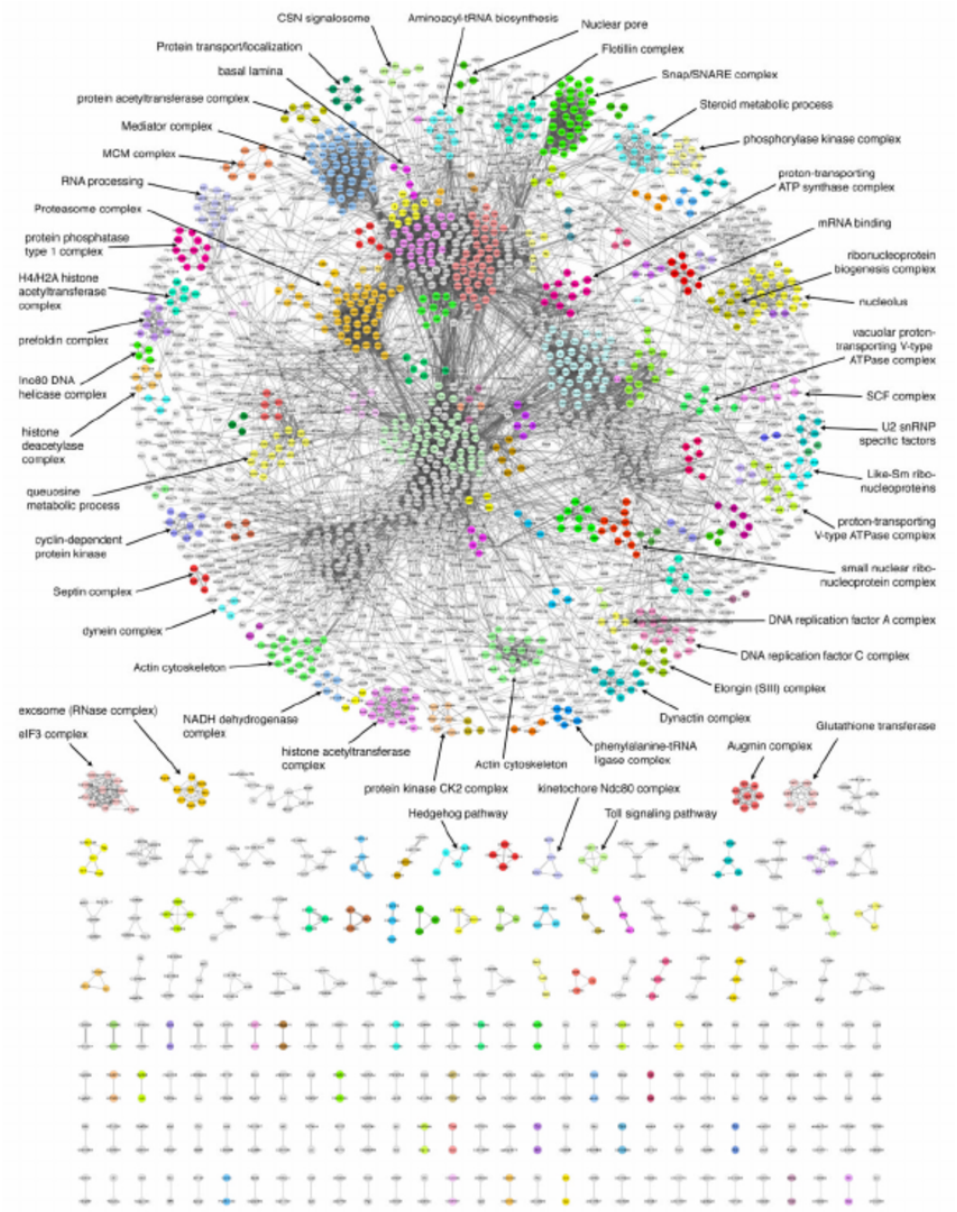
- The internet
  - Size 10^{13}



Opte project (https://en.wikipedia.org/wiki/Opte_Project)

Steven T. Smith

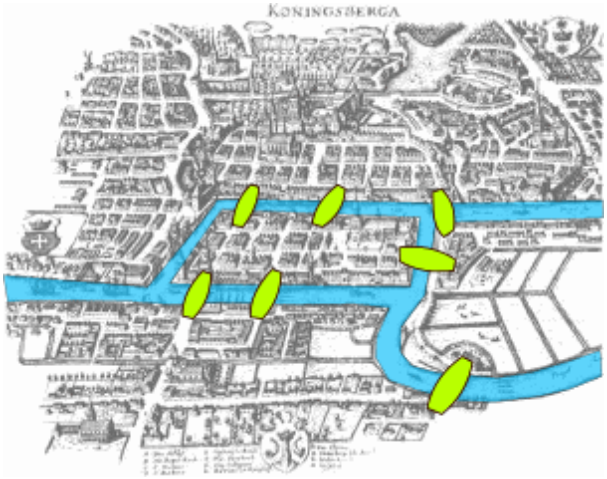
- Biology
  - *Drosphelia* protein interaction map
  - Size 10^4



Cell **147**(3):690-703 · October 2011 (https://www.ncbi.nlm.nih.gov/pubmed/22036573)

Steven T. Smith

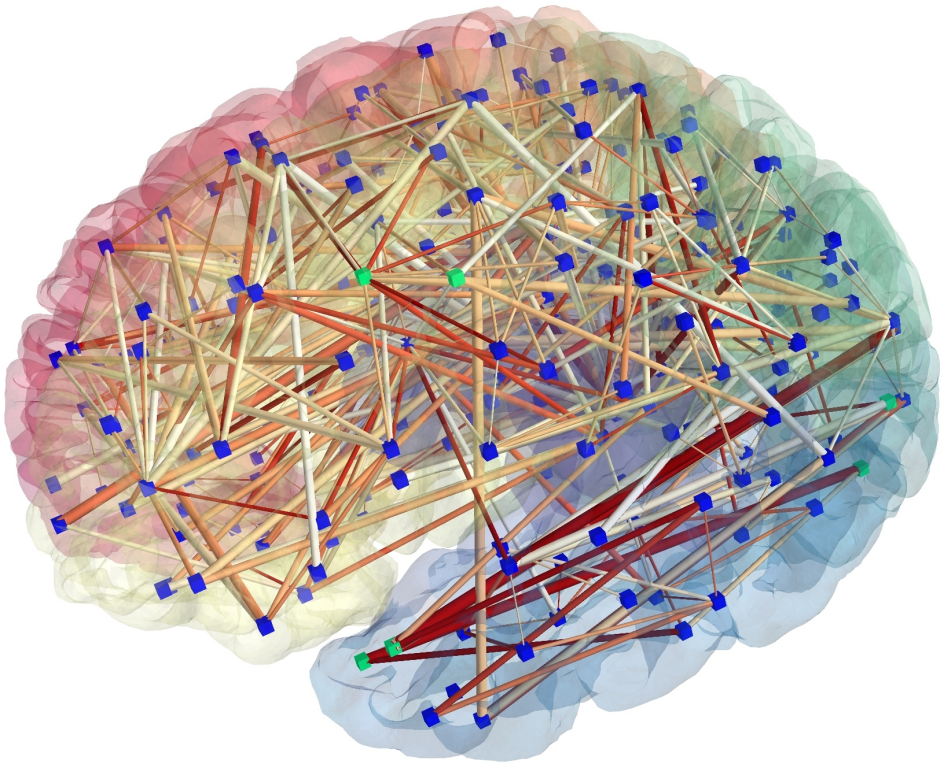
- Mathematics
  - Euler's bridges of Königsberg problem
  - Size 7
  - Invention of graph theory



Bridges of Königsberg (https://en.wikipedia.org/wiki/Seven_Bridges_of_Königsberg)

Steven T. Smith

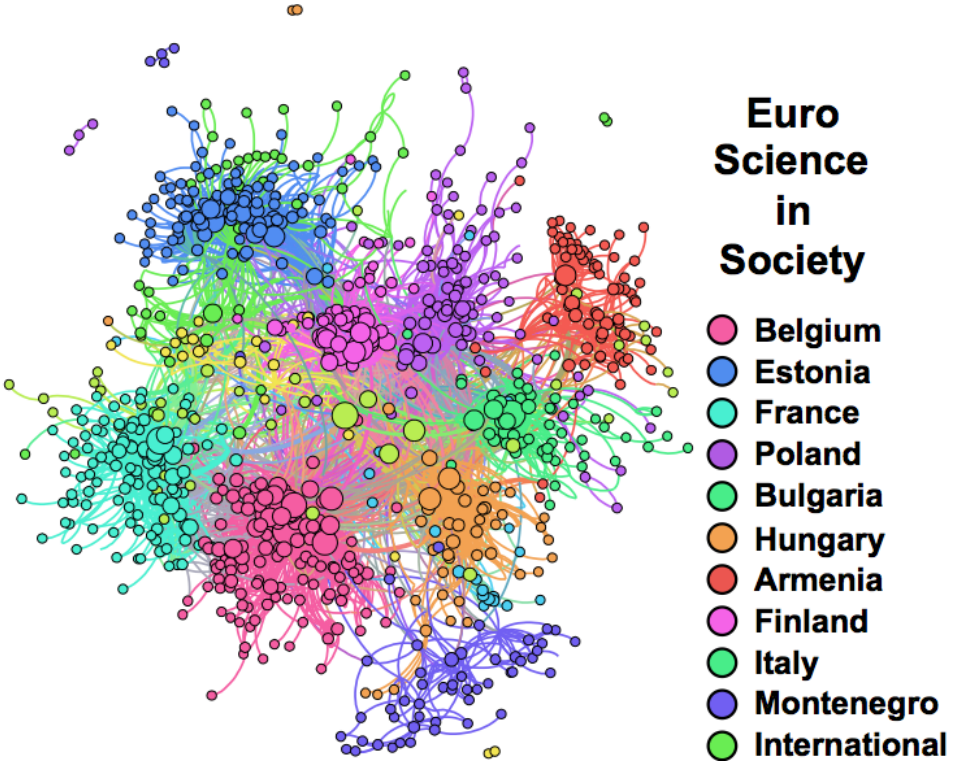
- Neurology
  - Human brain connectome
  - Size 10^{14}



Human brain connectome, P. Hagmann, ECR · 2014 (http://blog.myesr.org/mri-reveals-the-human-connectome/)

Steven T. Smith

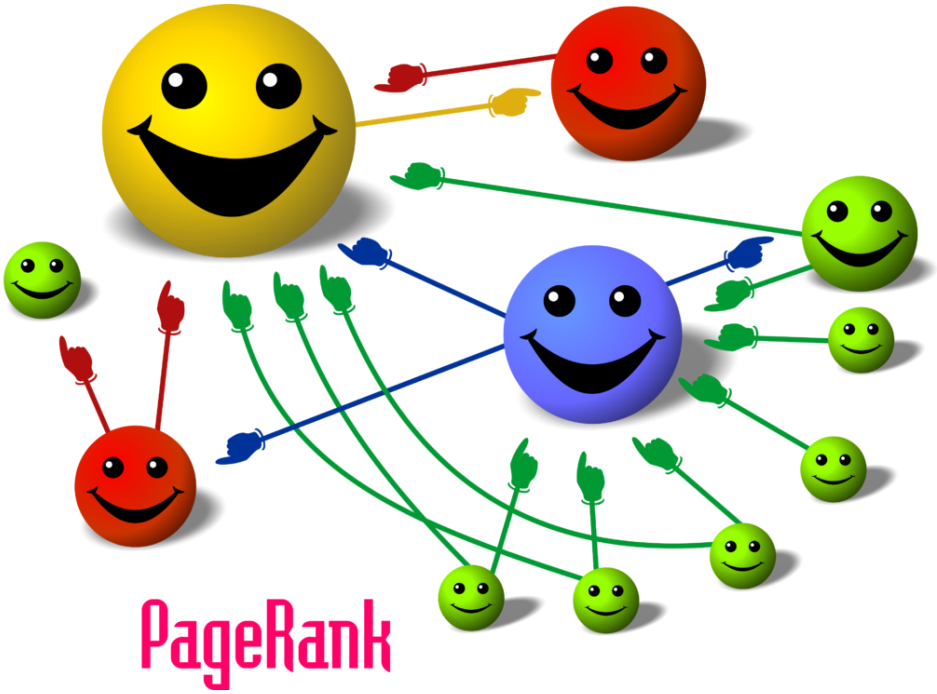
- Social Media
  - Scientific publications
  - Size 10^8



EuroSiS web mapping study (https://github.com/gephi/gephi/wiki/Datasets)

Steven T. Smith

- Information search
  - Google PageRank
  - Size 10^{13}



Google PageRank (https://en.wikipedia.org/wiki/PageRank)

MIT Courses with graph applications (it's all of them)

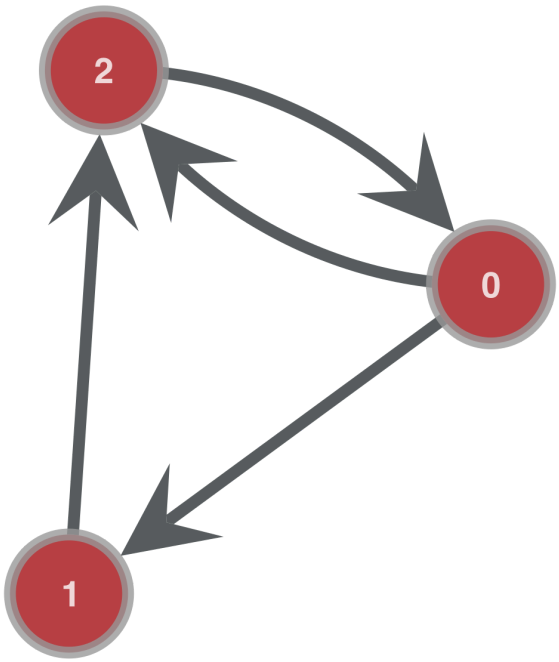
- Courses 1, 11: Road networks, traffic flows, social networks
- Courses 2, 3, 4, 16: Finite element methods
- Courses 5, 10, 12: Molecules, crystals, polymers, percolation, …
- Course 6: Circuits, algorithms, data analytics, graphics, cyber, …
- Courses 7, 9, 20: Molecules, protein interaction, neurology, population dynamics, …
- Course 8: QFT, crystals, polymers, complex networks, …
- Courses 14, 15, 17: Financial networks, social networks, game theory
- Course 18: Simplicial homology, numerical analysis, graph theory
- Course 21: Textual analytics, digital humanities
  - The first graph I ever saw was my middle school teacher's character graph (https://encrypted.google.com/search?q=character+graph+tale+of+two+cities) of _A Tale of Two Cities_

Steven T. Smith

Graph representations

Like every other kind of object, the design choice of how to represent graphs depends on the amount of data, kind of data, and algorithms that will be performed on the data. The data structure will help or hurt.

Here's a simple directed graph:



Steven T. Smith

Graph representation options

- "Triplestore" edge list (typically omit the `from:to` unless we're talking to a SQL database)

```
from:to 0 1
from:to 1 2
from:to 2 0
from:to 0 2
```

- Adjacency matrix $\mathbf{A}$; columns are "`from`", rows are "`to`" here (sparse format—no $O(V^2)$ storage!)

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 1 \\ 1 & 0 & 0 \\ 1 & 1 & 0 \end{pmatrix}$$

```
gt.adjacency(g)  # Sparse matrix example
<3x3 sparse matrix of type '<type 'numpy.float64'>'
    with 4 stored elements in Compressed Sparse Row forma
t>
```

Steven T. Smith

- Incidence matrix $\mathbf{B}$; columns are edges, rows are vertices (sparse format—no $O(VE)$ storage!)
  - I've never seen this used for graph representations, but it is really important in algebraic graph theory

$$\mathbf{B} = \begin{pmatrix} -1 & 0 & 1 & -1 \\ 1 & -1 & 0 & 0 \\ 0 & 1 & -1 & 1 \end{pmatrix}$$

- Adjacency lists
  - Store the neighbors of each vertex
  - "List" of "lists" (iterable of iterables)
  - Implicitly sparse
  - Cost of storage, lookup, insertions, deletions, and data re-use all depend on the underlying data structure used
  - Often convenient to store the data ***and*** its transpose

```
out_nbrs = [ [1,2], [2], [0] ]
in_nbrs = [ [2], [0], [0,1] ]
```

Steven T. Smith

- class Graph, Vertex, and Edge objects
  - The most practical and convenient in general situations
  - C++ Boost Graph Library (BGL), Python `graph-tool`, `igraph`, Java JUNG, …
- Dedicated graph database
  - Distributed and scalable
  - Neo4j, Titan, …
- Graph storage
  - Spreadsheets, CSV, XML, GML, GXL, GraphML, GEXF, `graphviz dot`, `.gt.bz2`, …

BFS algorithm

Problem: Find the distances between one vertex and all the rest.

Steven T. Smith

Breadth-first search solution:
Start at v_0

- distance[v_0] := 0
- For all neighbors of v_0:
  - distance[neighbors of v_0] := 1
  - For all neighbors of neighbors of v_0:
    - distance[neighbors of neighbors of v_0] := 2
    - And so on …

Maine memes (http://mainememes.com/2014/01/ya-cant-get-there-from-here/)

And add logic so you don't loop back to previously visited neighbors. This is also known as "snowball sampling."

The un-visited neighbors at each step are the "boundary" to be explored next.

BFS is a widely known algorithm.

Steven T. Smith

In [16]:
```
YouTubeVideo('JA7CKvoKEmE')
```

Out[16]:

Ignore this Erdős–Rényi random graph and plotting setup

In [17]:
```
N = 12
p = 1.5 * np.log(N)/N

g = gt.random_graph(N, deg_sampler=lambda: npr.poisson((N-1)*p),
    directed=False, model='erdos')
vroot = g.vertex(0)
```

Steven T. Smith

In [18]:
```
pos = gt.sfdp_layout(g,C=0.5)

# graph plotting properties
vertex_text = g.new_vertex_property('string')
vertex_fill_color = g.new_vertex_property('vector<float>')
vertex_size = g.new_vertex_property('float')

# a little chrome
not_reached_color = [0.8, 0.8, 1., 0.9]
reached_color = [0.640625, 0, 0, 0.9]
not_reached_size = 36
reached_size = 48
bfs_figure = {}
def int2spreadsheet(n,b=string.ascii_uppercase):
    d, m = divmod(n,len(b))
    return int2spreadsheet(d-1,b)+b[m] if d else b[m]
for v in g.vertices():
    vertex_text[v] = int2spreadsheet(g.vertex_index[v])
def hop_init_graphics():
```

Steven T. Smith

```
    for v in g.vertices():
        vertex_text[v] = int2spreadsheet(g.vertex_index[v])
    vertex_fill_color.set_value(not_reached_color)
    vertex_fill_color[vroot] = reached_color
    vertex_size.set_value(not_reached_size)
    vertex_size[vroot] = reached_size
def plot_graph_hop():
    res = gt.graph_draw(g, pos=pos,
                  vertex_text=vertex_text,
                  vertex_font_family='sans-serif', vertex_font_size=18,
                  vertex_font_weight=cairo.FONT_WEIGHT_BOLD,
                  vertex_fill_color=vertex_fill_color,
                  vertex_size=vertex_size,
                  output_size=gd_output_size,
                  output=get_figure_name(),
                  inline=False)
# more chrome
def hop_plot_graphics():
    figure_text = '''Hop {:d}, Vertices: {}
    Boundary: {}'''.format(hop, \
        ', '.join([int2spreadsheet(g.vertex_index[v]) for v in g.vertices()
 if 0 <= bfs[v] and bfs[v] <= hop]), \
        ', '.join([int2spreadsheet(g.vertex_index[v]) for v in boundary]))
#     print('Hop {:d}, vertices {}:'.format(hop, \
#         ', '.join([int2spreadsheet(g.vertex_index[v]) for v in g.vertices
() if 0 <= bfs[v] and bfs[v] <= hop])))
#     print('Boundary {}:'.format(', '.join([int2spreadsheet(g.vertex_index
[v]) for v in boundary])))
    bfs_figure[hop] = {'text': figure_text, 'image': get_figure_name()}
    plot_graph_hop() # plot the results
    increment_figure_number()
def hop_update_graphics(u=vroot):    # plotting niceties
    vertex_text[u] += ' ({:d})'.format(bfs[u])
    vertex_fill_color[u] = vertex_fill_color[vroot]
    vertex_fill_color[u][3] = max(0.2,vertex_fill_color[vroot][3]-hop/8.)
    vertex_size[u] = vertex_size[vroot]
def hop_display(num=None):
    if num is not None:
        get_ipython().run_cell_magic(u'HTML', u'', \
            u'{}<p style="text-align:center;"><img src="{}"/>
</p>'.format(''.join(['<h4>'+l+'</h4>' for l in bfs_figure[num]['text'].spl
it('\n')]),bfs_figure[num]['image']))
```

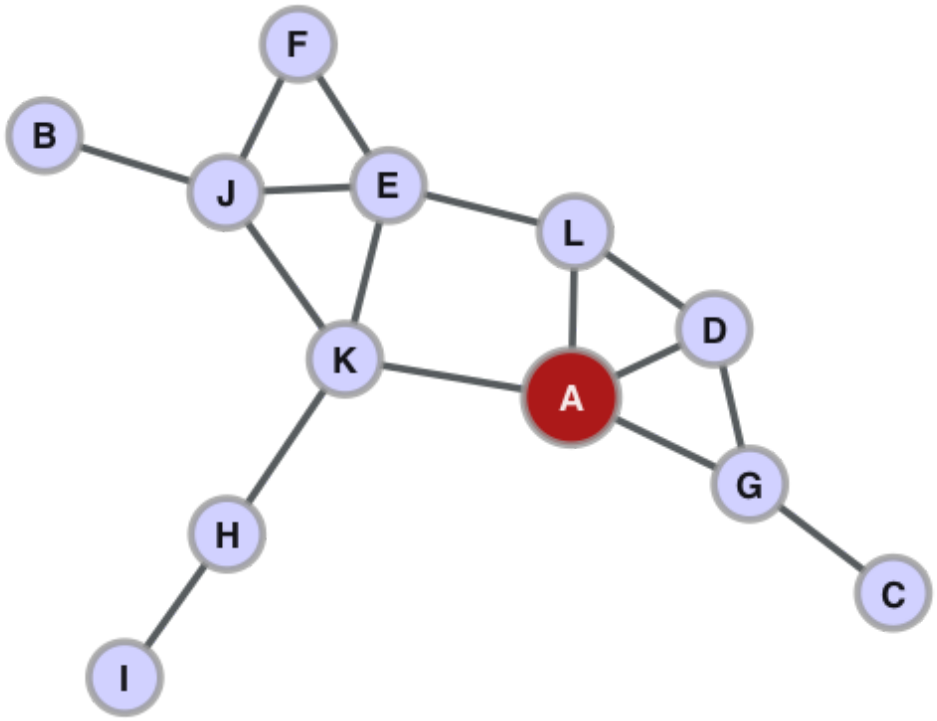
Back to the lecture!

Steven T. Smith

BFS example graph

We will run the BFS algorithm on this example random graph. The initial vertex 'A' is colored **red**.

In [19]:
```
hop_init_graphics()
plot_graph_hop()

display_figure_name()
```



/>

Steven T. Smith

Run the BFS algorithm

The BFS algorithm will now be run on the example graph.

After each step, the visited vertices are displayed in **red**, and the distance from the initial vertex 'A' is shown in parentheses.

The vertex property `bfs` (an integer) is used to store the distance between 'A' and all other vertices.

Before the algorithm has completed, negative distances imply that the vertex is un-visited.

After the algorithm completes, negative distances imply unreachability.

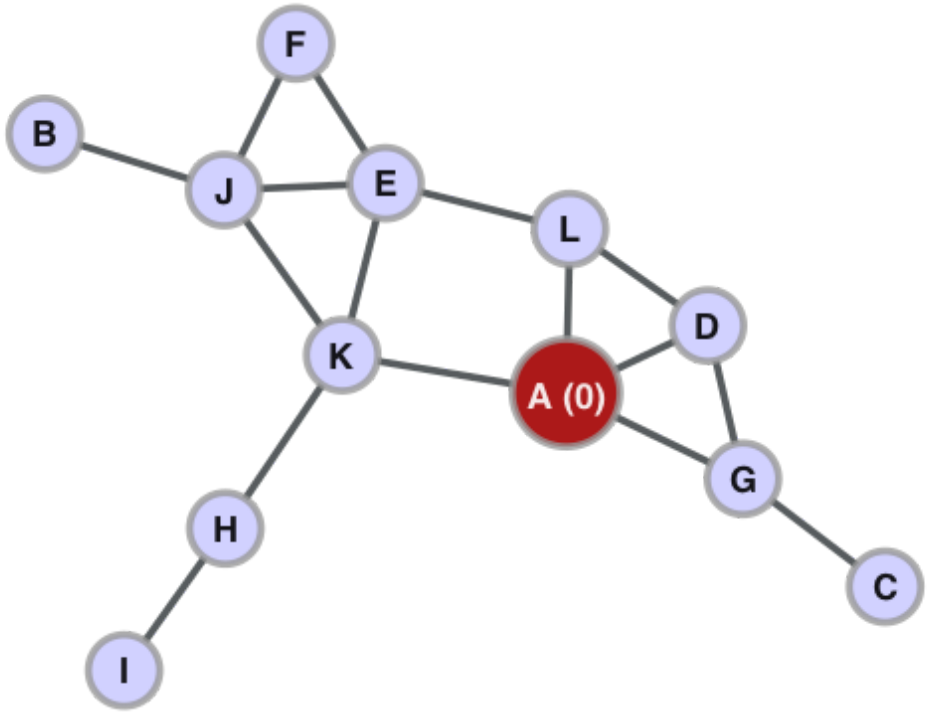
In [20]:
```
# bfs vertex property: >= 0 the BFS; < 0 == not reached
bfs = g.new_vertex_property('int')
bfs.set_value(-1)

# initialization
hop = 0
bfs[vroot] = hop
hop_init_graphics()
hop_update_graphics()

# BFS algorithm
boundary = set([ vroot ]) # start exploring at vroot
while len(boundary) > 0:
    hop_plot_graphics()
    hop += 1
    boundary_next = set() # the next boundary as we explore
    for v in boundary:
        for u in v.all_neighbours():
            if bfs[u] < 0 and u not in boundary_next:
                # add to the boundary if previously unvisited
                boundary_next.add(u)
                bfs[u] = hop  # save the distance
                hop_update_graphics(u)
    boundary = boundary_next
# the algorithm terminates when it runs out of neighbors
```

Steven T. Smith

In [21]:
```
if hop-1 >= 0: hop_display(0)
```

Hop 0, Vertices: A

Boundary: A



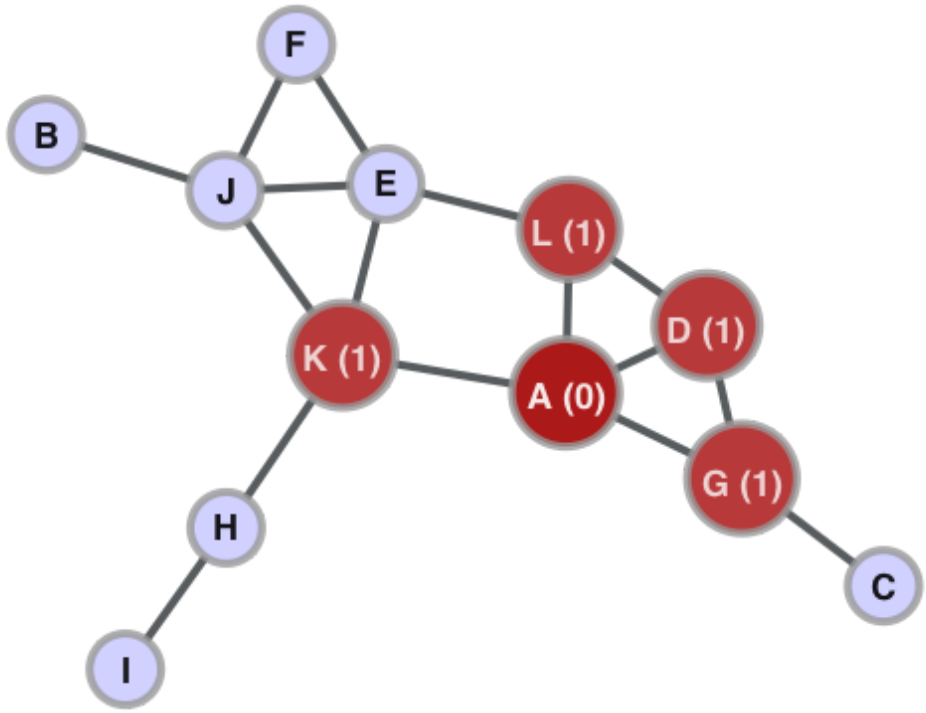
/>

Steven T. Smith

In [22]:
```
if hop-1 >= 1: hop_display(1)
```

Hop 1, Vertices: A, D, G, K, L

Boundary: D, K, L, G



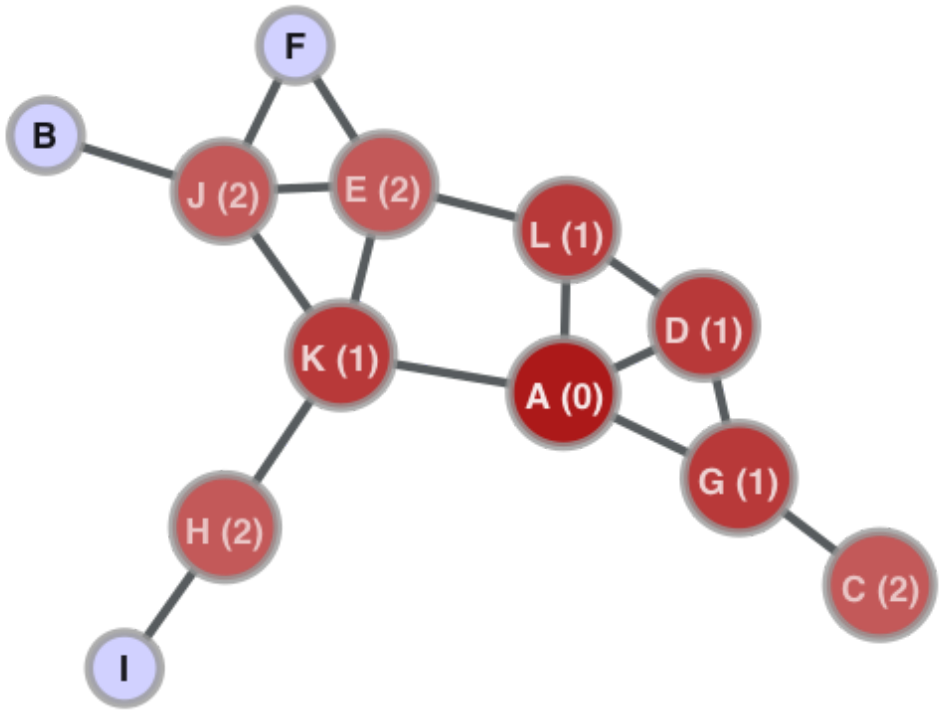
/>

Steven T. Smith

In [23]:
```
if hop-1 >= 2: hop_display(2)
```

Hop 2, Vertices: A, C, D, E, G, H, J, K, L

Boundary: J, C, E, H



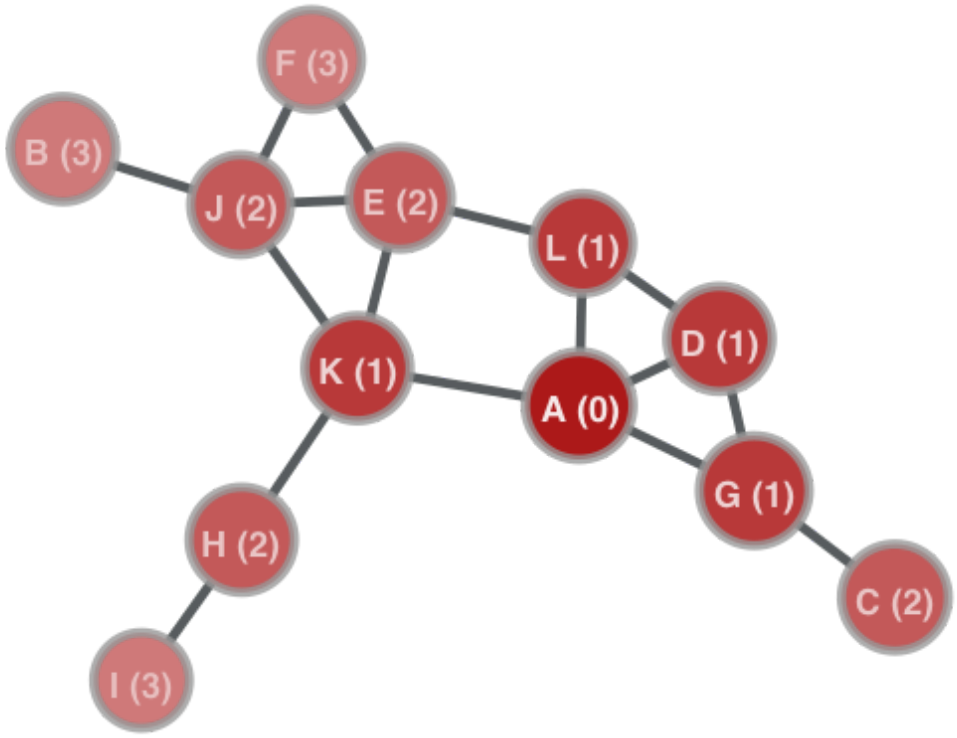
/>

Steven T. Smith

In [24]:
```
if hop-1 >= 3: hop_display(3)
```

Hop 3, Vertices: A, B, C, D, E, F, G, H, I, J, K, L

Boundary: I, B, F



/>

In [25]:
```
# plot the rest
for h in range(4,hop): hop_display(h)
```

Steven T. Smith

Algorithm analysis

The main work of the BFS algorithms takes place within the following `for`-loop. By construction, we only access each vertex once, and each edge once; therefore, the BFS algorithm has $O(V + E)$ linear complexity, which is what we would hope for.

```
for v in boundary:
        for u in v.all_neighbours():
```

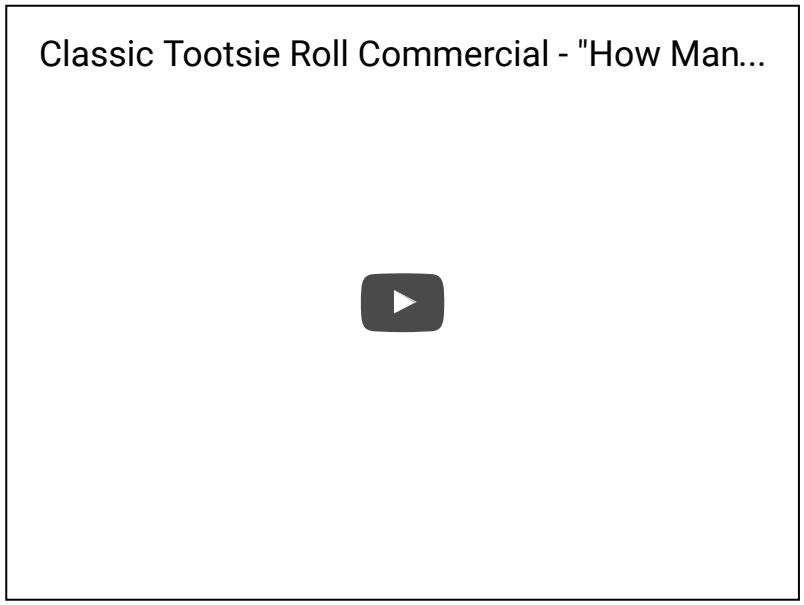
Use math to validate the algorithm's behavior

- The number of BFS hops between to the root vertex and any other vertex is the shortest path ("geodesic") length between the root and the other vertex. By construction, if there were a shorter path, BFS would have already found it.
- The ***diameter*** of a graph is the length of the longest minimal path ("geodesic") within the graph. The diameter of our example graph above is {{int(gt.pseudo_diameter(g)[0])}}.
- On average, how many iterations are required for BFS?

Steven T. Smith

In [26]:
```
YouTubeVideo('O6rHeD5x2tI',start=35)
```

Out[26]:



Cool fact #1†: The ***diameter*** of an Erdős–Rényi graph is almost surely,

$$\operatorname{diam}(\mathrm{ER}) \overset{\text{a.s.}}{\to} \left\lceil \frac{\log N}{\log pN} \right\rceil \quad \text{as} \quad pN \to \infty.$$

†Fan Chung and Linyuan Lu, "The Diameter of Sparse Random Graphs," *Advances in Applied Mathematics* **26**:257–279 (2001).

In [27]:
```
diam = int(np.ceil((np.log(N))/np.log(p*N)))
print('The diameter of these examples is almost surely {:d}.'.format(diam))
```

```
The diameter of these examples is almost surely 2.
```

If you re-run this notebook several times, you'll see that BFS typically terminates after {{diam}} hops for Erdős–Rényi graphs of order {{N}} and $p = $ {{'{:.3f}'.format(p)}}.

Steven T. Smith

Cool fact #2‡: The average path length of an Erdős–Rényi graph equals,

$$l(\mathrm{ER}) = \frac{\log N - \gamma}{\log pN} + \frac{1}{2},$$

where $\gamma = 0.5772\ldots$ is the Euler-Mascheroni constant.

‡A. Fronczak, P. Fronczak, and J. A. Hołyst, "Average path length in random networks," *Phys. Rev. E* **70**:056110 (2004).

In [28]:
```
lave = (np.log(N)-np.euler_gamma)/np.log(p*N)+0.5
print('The average path length of these examples is {0:.3f}.'.format(lave))
```

```
The average path length of these examples is 1.950.
```

We'd have to run BFS from every vertex to collect the statistics to check this result. This will be a subject of future lectures …

Markov Chains and Perron-Frobenius

Markov chains

A graph whose edges have probabilities represents a state transition or "Markov" process.

Here's a simple example from before:

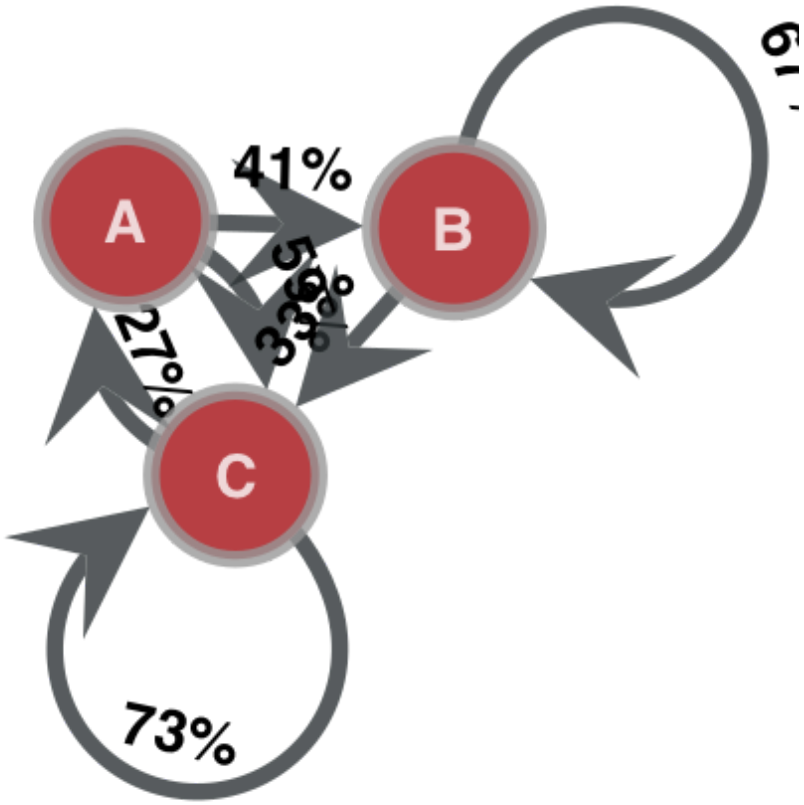
Steven T. Smith

In [29]:
```
g_markov = gt.Graph()
g_markov.add_edge_list(edge_list + [edge_list[-1][::-1],(1,1),(2,2)])
g_markov.set_directed(True)
edge_weight_markov = g_markov.new_edge_property('float')
# Markov chain transition weights
for e in g_markov.edges():
    edge_weight_markov[e] = npr.uniform()
T = gt.transition(g_markov,weight=edge_weight_markov)
for e in g_markov.edges():  # normalize to simplex
    edge_weight_markov[e] = T[g_markov.vertex_index[e.target()],g_markov.ve
rtex_index[e.source()]]
```

In [30]:
```
edge_text_markov = g_markov.new_edge_property('string')
for e in g_markov.edges():
    edge_text_markov[e] = '{:d}%'.format(int(np.round(100.*T[g_markov.verte
x_index[e.target()],g_markov.vertex_index[e.source()]])))
for v in g.vertices():
    vertex_text[v] = int2spreadsheet(g.vertex_index[v])
pos_markov = gt.sfdp_layout(g_markov)
res = gt.graph_draw(g_markov, pos=pos_markov,
              vertex_text=vertex_text,
              vertex_font_size=30,
              vertex_font_family='sans-serif',
              vertex_font_weight=cairo.FONT_WEIGHT_BOLD,
              edge_text=edge_text_markov,
              edge_font_size=30,
              edge_font_family='sans-serif',
              edge_font_weight=cairo.FONT_WEIGHT_BOLD,
              edge_text_distance=18,
              output_size=gd_output_size,
              output=get_figure_name(), inline=False, fit_view=.5)
```

Steven T. Smith

In [31]:
```
display_figure_name()
```



/>

Example Markov process

If the Markov process starts at state 'A', then the probability of jumping to the neighbors of 'A' is given by the edge probabilities.

Here's a Markov process:

Steven T. Smith

In [32]:
```
state = g_markov.vertex(0)
markov_process = int2spreadsheet(g_markov.vertex_index[state])
number_of_jumps = 40
for k in range(number_of_jumps):
    edges = [e for e in state.out_edges()]
    next = int(np.nonzero(npr.multinomial(1, [edge_weight_markov[e] for e i
n edges], size=1)[0])[0][0])
    state = edges[next].target()
    markov_process += int2spreadsheet(g_markov.vertex_index[state])
print 'Markov process: {}…'.format(markov_process)
```

```
Markov process: ACCCCABBBCABCCCCCCCCCCCCABBBBCACCCCABBBC
C…
```

Adjacency and transition matrix conventions

- The adjacency matrix is defined $A_{ij} = 1$ iff the edge (v_i, v_j) exists. The out-degree vector is equal to $\mathbf{d} = \mathbf{A}\cdot\mathbf{1}$.
- The (right) stochastic transition matrix is a matrix $\mathbf{T} = (t_{ij})$ with t_{ij} denoting the probability of jumping from v_i to v_j. Therefore, the row sum of $\mathbf{T}$ is unity: $\mathbf{T}\cdot\mathbf{1} = \mathbf{1}$. The rows of $\mathbf{T}$ represent the vector of probabilities of transition, and the a probability distribution vector $\mathbf{p}_0$ on the graph transitions as $\mathbf{p}_1^{\mathrm{T}} = \mathbf{p}_0^{\mathrm{T}}\mathbf{T}$.

- Many authors and packages (e.g. `graph-tool`) use the ***opposite, transposed*** convention for adjacency matrices, in which the out-degree vector is determined by $\mathbf{d} = \mathbf{A}^{\mathrm{T}}\cdot\mathbf{1}$, and $\mathbf{T}$ is a (left) stochastic transition matrix whose column sums are unity, $\mathbf{T}^{\mathrm{T}}\cdot\mathbf{1} = \mathbf{1}$. The code in this notebook will use the `graph-tool` convention, which is illustrated by the example below.

Our example transition matrix looks like this:

Steven T. Smith

In [33]:
```
T = gt.transition(g_markov,weight=edge_weight_markov)

print('Transition matrix of a directed graph (not symmetric!)')
print(T.todense())

print('\n')
print('Column sum of a transition matrix (all ones!)')
print np.squeeze(np.array(T.sum(0)))    # should be all ones
```

```
Transition matrix of a directed graph (not symmetric!)
[[ 0.          0.          0.27262138]
 [ 0.41431689  0.66766667  0.        ]
 [ 0.58568311  0.33233333  0.72737862]]


Column sum of a transition matrix (all ones!)
[ 1.  1.  1.]
```

The stationary distribution

What fraction of time does a Markov process spend in state 'A'? State 'B'? And so forth …

- The probability vector for state 'A' looks like

$$p_{\mathrm{A}} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

- After one jump, the probability equals $\mathbf{T} \cdot p_{\mathrm{A}}$
- After two jumps, the probability equals $\mathbf{T} \cdot \mathbf{T} \cdot p_{\mathrm{A}} = \mathbf{T}^2 \cdot p_{\mathrm{A}}$
- … After k jumps, $\mathbf{T}^k \cdot p_{\mathrm{A}}$

Steven T. Smith

- The ***stationary distribution*** is achieved when
$$\mathbf{T}\cdot\left(\mathbf{T}^k\cdot p_{\mathrm{A}}\right) = \mathbf{T}^k\cdot p_{\mathrm{A}}$$
- Call this distribution p_1 because the stationarity condition
$$\mathbf{T}\cdot p_1 = p_1$$
implies that p_1 is an eignevector of $\mathbf{T}$ with corresponding eigenvalue 1.

Perron-Frobenius Theorem

Theorem: The transition matrix of a *strongly connected* graph has maximal eigenvalue 1. This eigenvalue is simple (one-dimensional invariant subspace), and the corresponding eigenvector p_1 satisfies $p_1 > 0$.

- Scale p_1 so that $\sum_i (p_1)_i = 1$
- The transition matrix $\mathbf{T}$ of a strongly connected graph is a ***nonnegative***, ***irreducible***[*] matrix.

[*]A reducible matrix $\mathbf{T}$ has states (invariant subspaces) of dimension smaller than the range of the matrix, and $\mathbf{T}$ looks like
$$\mathbf{T} = \begin{pmatrix} \mathbf{Q} & \mathbf{R} \\ \mathbf{0} & \mathbf{I} \end{pmatrix}$$

Steven T. Smith

In [34]:
```
one,p1 = (r.real for r in sparse.linalg.eigs(T, k=1, which='LM'))
p1 = p1/p1.sum()

print('The principal eigenvalue is {}.'.format(one))
print('The principal eigenvector is\n{}.'.format(p1))
print('The error of p1-T1*p1 is is\n{}.'.format(p1-T*p1))
```

```
The principal eigenvalue is [ 1.].
The principal eigenvector is
[[ 0.16906795]
 [ 0.21077545]
 [ 0.6201566 ]].
The error of p1-T1*p1 is is
[[ -2.77555756e-17]
 [ -2.77555756e-17]
 [  0.00000000e+00]].
```

The cells below load the style of this notebook, along with window scolling embiggening (http://stackoverflow.com/questions/18770504/resize-ipython-notebook-output-window).

In [39]:
```
# ***This cell cannot be set as "Skip" in Slideshow, or
# `reveal.js` will not load the css file.***

# Execute this cell to load the notebook's style sheet, then ignore it
HTML(open('styles/numericalmoocstyle_sts.css','r').read())
```

Out[39]:

In [38]:
```
display(HTML('''
<style>
.output_wrapper, .output {
    height:auto !important;
    max-height:8096px;  /* your desired max-height here */
}
.output_scroll {
    box-shadow:none !important;
    webkit-box-shadow:none !important;
}
</style>
'''))
```

Steven T. Smith