Graph Data Visualization

Milan Vojnovic

ST445 Managing and Visualizing Data

Scope of this lecture

- Learn about how to manipulate and visualize graph data
- Learn about different graph layouts and their principles
- Focus on graph layouts implemented in NetworkX and Graphviz

Basic definitions

- A graph G consists of a set V of vertices and a set E of edges
- Standard notation: $G = (V, E)$
- An edge $e = (u, v) \in E$ indicates a relationship between vertices u and v
- An undirected graph is a graph where edges have no direction: each edge is an unordered pair of vertices
- Both vertices and edges may have attributes

Graph examples

- Social graphs: representing people and their social interactions
  - Ex. Facebook, Linkedin, email exchanges, …
- Knowledge graphs: representing entities and their relationships
  - Ex. Google knowledge graph, Bing Satori, Freebase, Yago, Wordnet, …
- Collaboration graphs: representing people and their collaborations
  - Ex. Co-authorships from dblp, Google Scholar, Microsoft Academic search, …
- Product purchase graphs: representing who bought what
  - Ex. Amazon product purchases
- Product reviews: ratings of products or services provided by users
  - Ex. TripAdvisor
- Infrastructure graphs: road networks, communication networks, …

Our tree recurrent examples

- GitHub organizations:

  co-activities in GitHub repositories belonging to different organizations

- GitHub programming languages:

  co-use of different programming languages in GitHub repositories

- LANL routes:

  round-trip time measurements between network nodes of a local area network

Why visualizing graphs?

- For exploratory data analysis by visual inspection of graph drawings

- For visual detection of graph structures
  - Ex. community detection in social networks
  - Ex. tree structures
  - Ex. feed-forward structures

- For communicating about graph properties

- For making ”cool” plots

Graph visualization using Matplotlib

- networkX: a Python package for the creation, manipulation, and study of the structure of networks
  - https://github.com/networkx

- Graphviz (*Graph Visualization Software*): *a* package of open-source tools initiated by AT&T Labs Research for drawing graphs specified in DOT language scripts
  - http://www.graphviz.org, https://www2.graphviz.org

- Pygraphviz: a Python interface to the Graphviz graph layout and visualization package
  - https://pygraphviz.github.io

Other graph visualization libraries

- Cytoscape: http://www.cytoscape.org
- Gephi: https://gephi.org
- Tensorflow: https://www.tensorflow.org/versions/r0.9/how_tos/graph_viz/index.html
- Caffe: https://github.com/BVLC/caffe/blob/master/python/caffe/draw.py

Useful resources

- NetworkX tutorial
  - https://networkx.github.io/documentation/networkx-1.10/tutorial/index.html

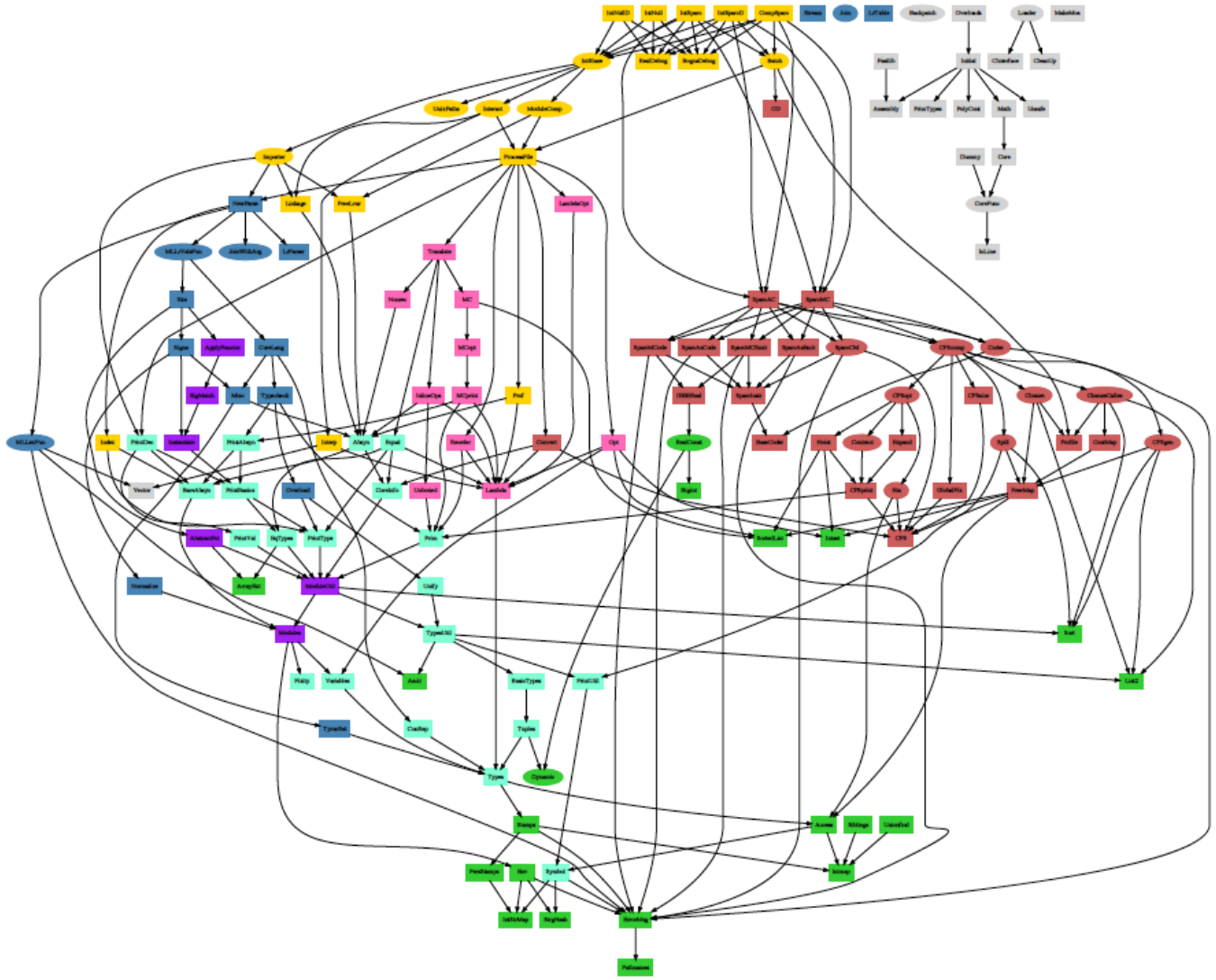
NetworkX graph drawing layouts

- Circular: position nodes on a circle
  - circular_layout(G[, scale, center, dim])
- Random: position nodes uniformly at random in the unit square
  - rescale_layout(pos[, scale])
- Shell: position nodes in concentric circles
  - shell_layout(G[, nlist, scale, center, dim])
- Spring: position nodes using Fruchterman-Reingold force-directed algorithm
  - spring_layout(G[, k, pos, fixed, …])
- Spectral: position nodes using the eigenvectors of the graph Laplacian
  - spectral_layout(G[, weight, scale, center, dim])

Graphviz graph drawing layouts

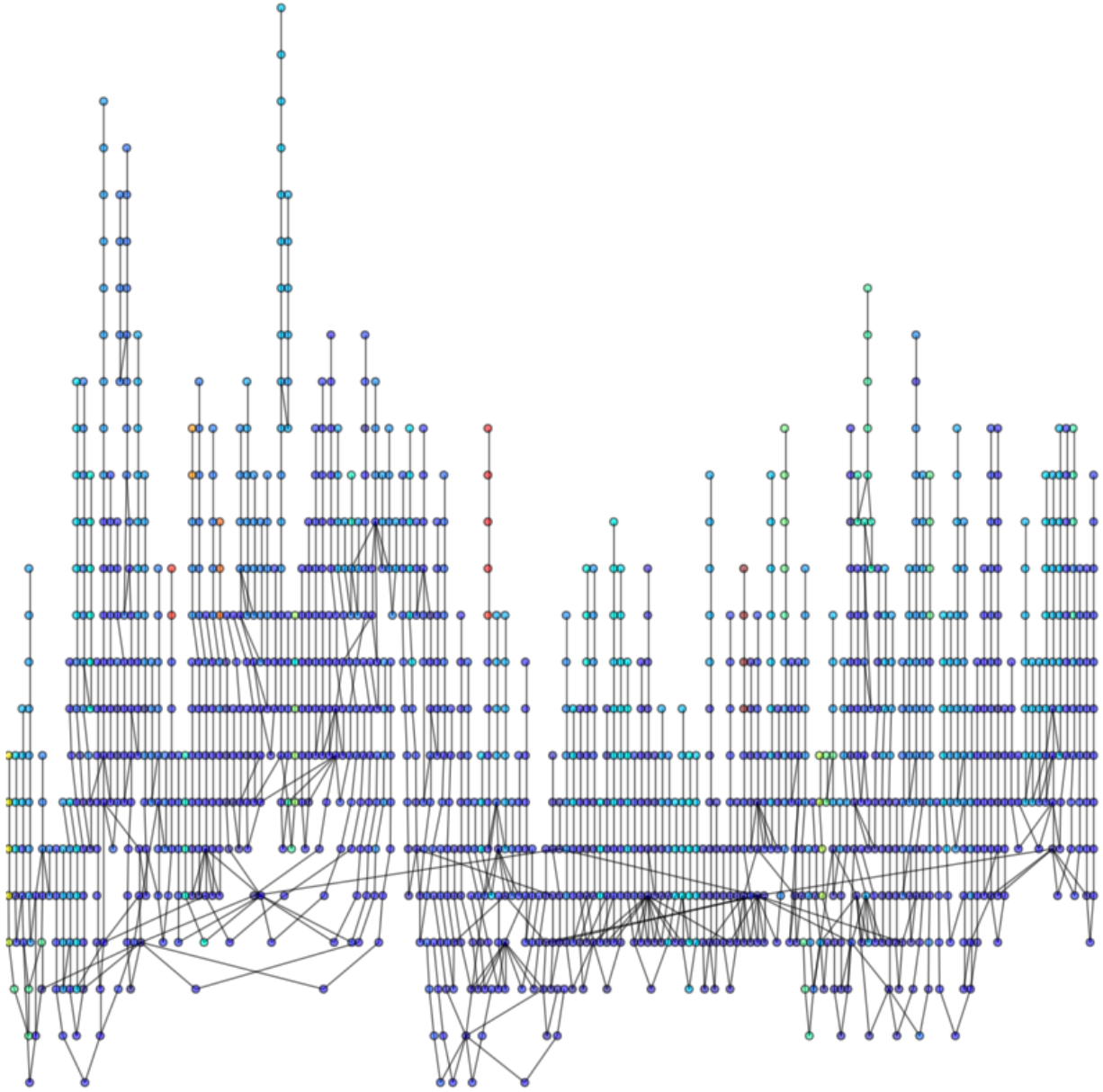
- dot: graphs that can be drawn as hierarchies or have a natural “flow”
- neato: spring model for undirected graphs by Kamada and Kawai (1989)
- twopi: radial layouts Wills (1997)
- circo: circular layout, Six and Tollis (1999) and Kaufmann and Wiese (2002)
- fdp: spring model for undirected graphs in the spirit of Fruchterman and Reingold (1991)
- sfdp: speed optimized version of fdp
- patchwork: squarified treemap, Brulls et al (2000)
- osage: draws a graph using its cluster structure

Graph layout: dot



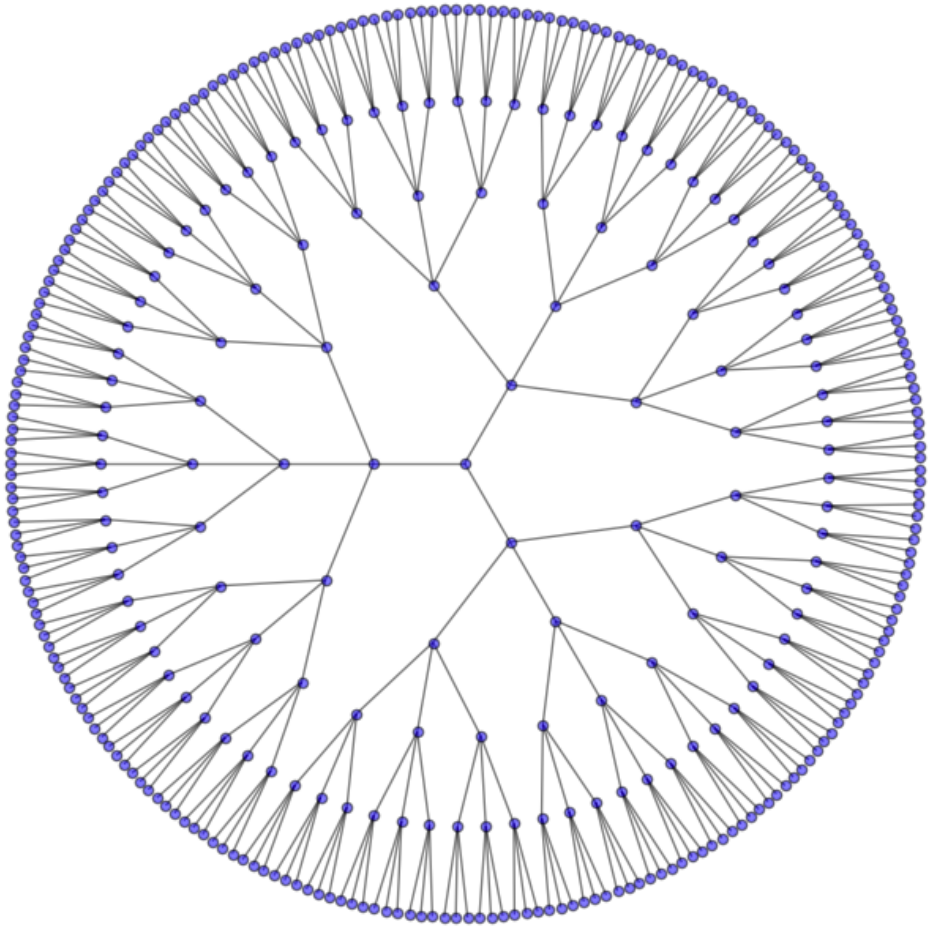
Graph layout: dot

- LANL routes example



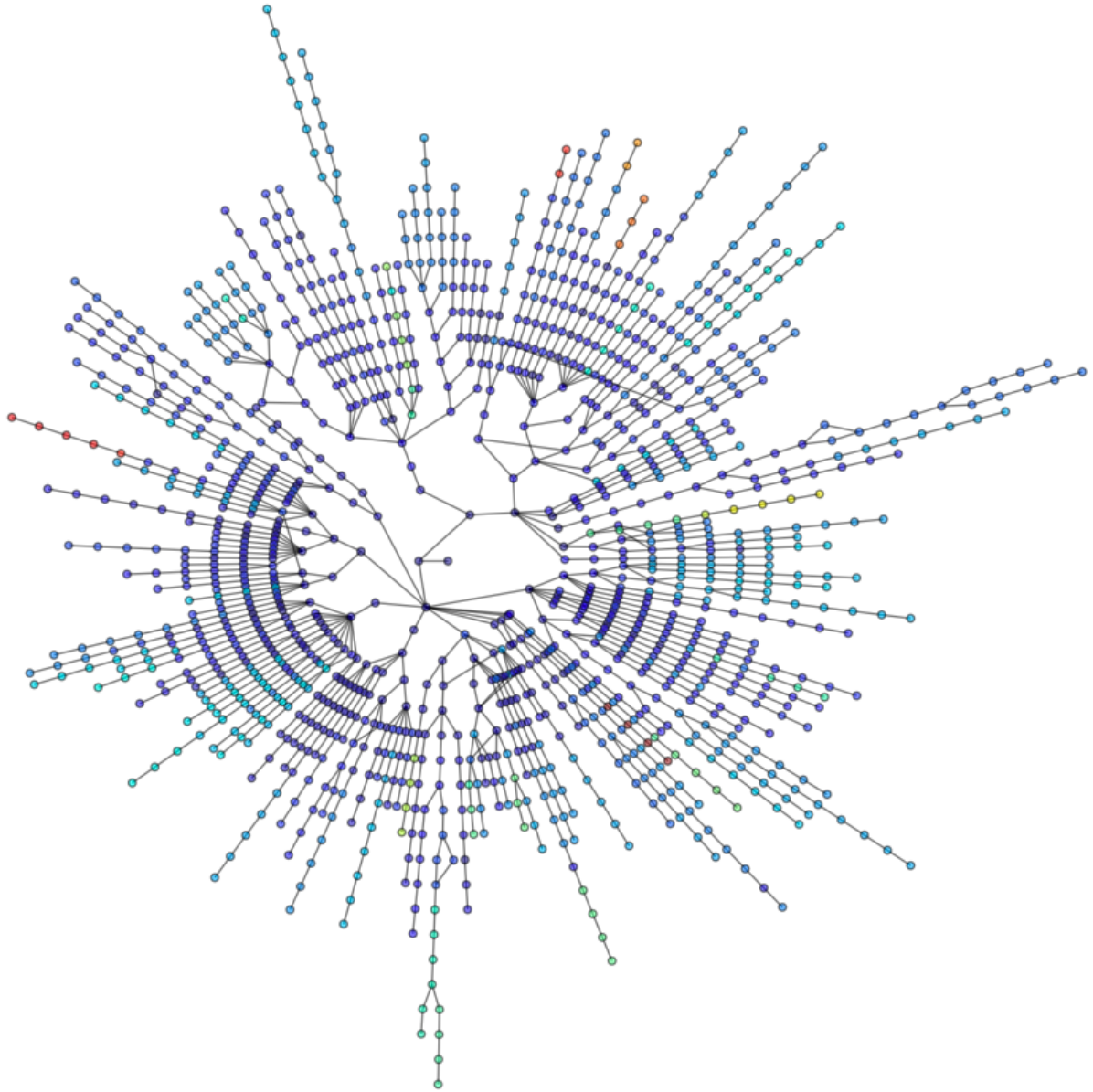
Graph layout: twopi

- A node is chosen as the center node and put at the origin
- The remaining nodes are placed on a sequence of concentric circles centered about the origin, each at a fixed radial distance from the previous circle
- All nodes at distance 1 from the center are placed on the first circle; all nodes distance 1 from a node on the first circle are placed on the second circle; and so forth



Graph layout: twopi

- LANL routes example

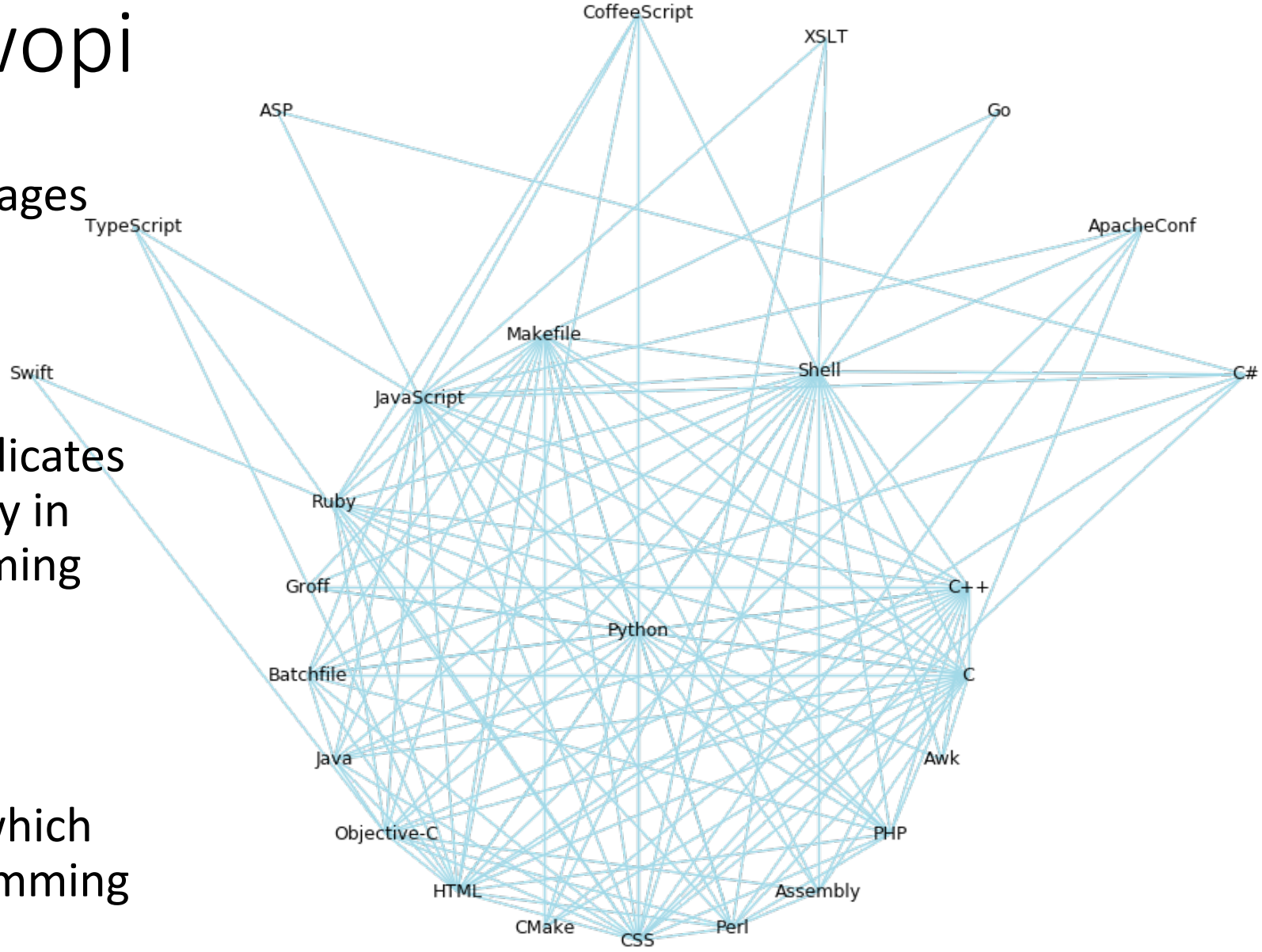


Graph layout: twopi

- GitHub organizations example
- An edge between two organizations indicates existence of an actor who initiated an activity (other than starring a repository) to at least one repository of each organization

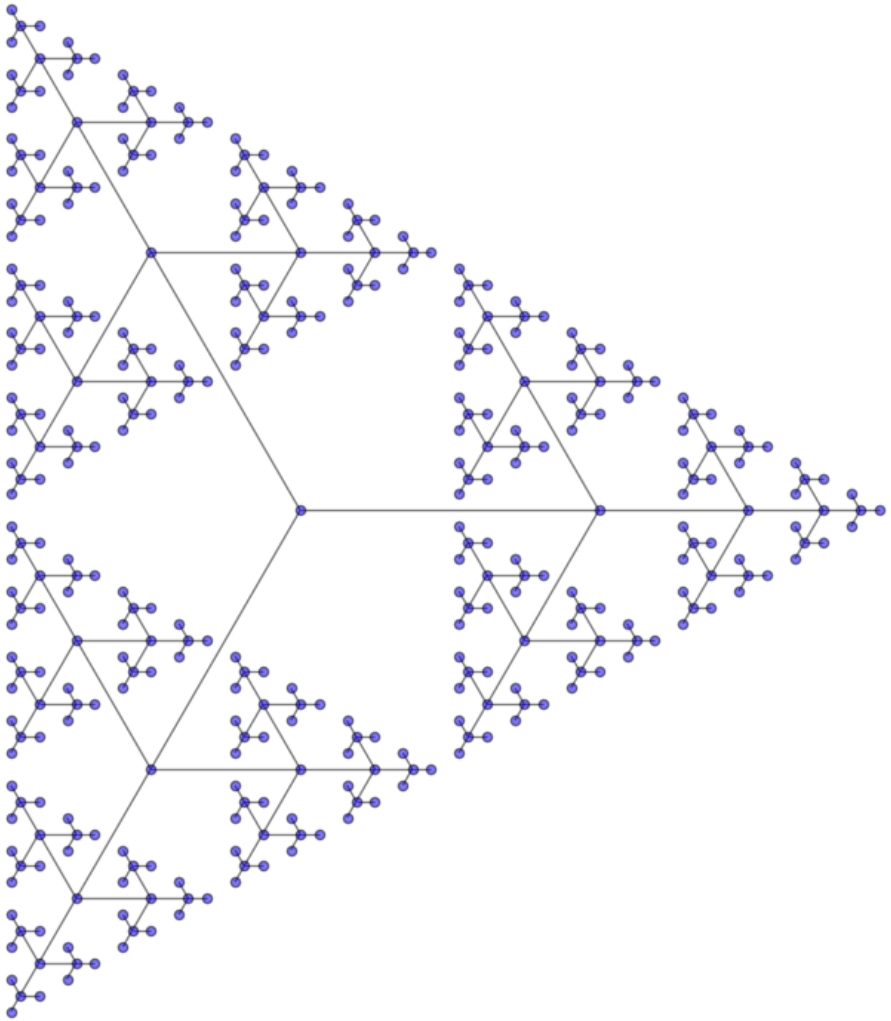
Graph layout: twopi

- GitHub programming languages example
- An edge between two programming languages indicates the existence of a repository in which both these programming languages are used
- Edge weight is equal to the number of repositories in which both corresponding programming languages are used



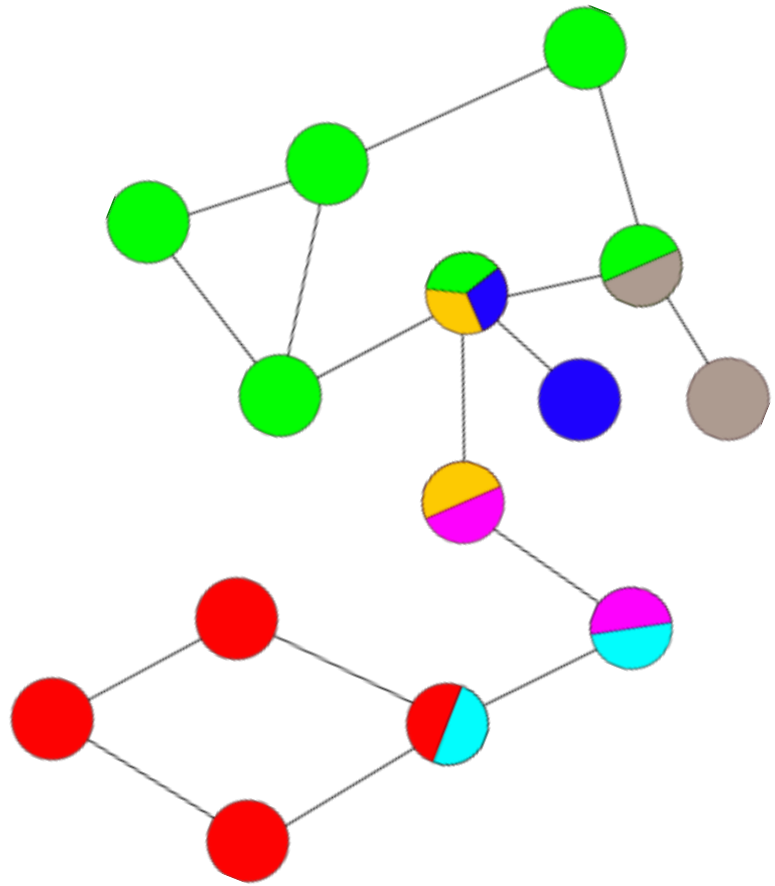
Graph layout: circo

- This method identifies bi-connected components and draws them along a circle
- The block cut-point tree is then laid out using a recursive radial algorithm
- Edge crossings within a circle are minimized by placing as many edges on the circle's perimeter as possible. If the component is outer planar, the component will have a planar layout
- If a node belongs to multiple non-trivial bi-connected components, the layout puts the node in one of them. By default, this is the first non-trivial component found in the search from the root component



Biconnected components

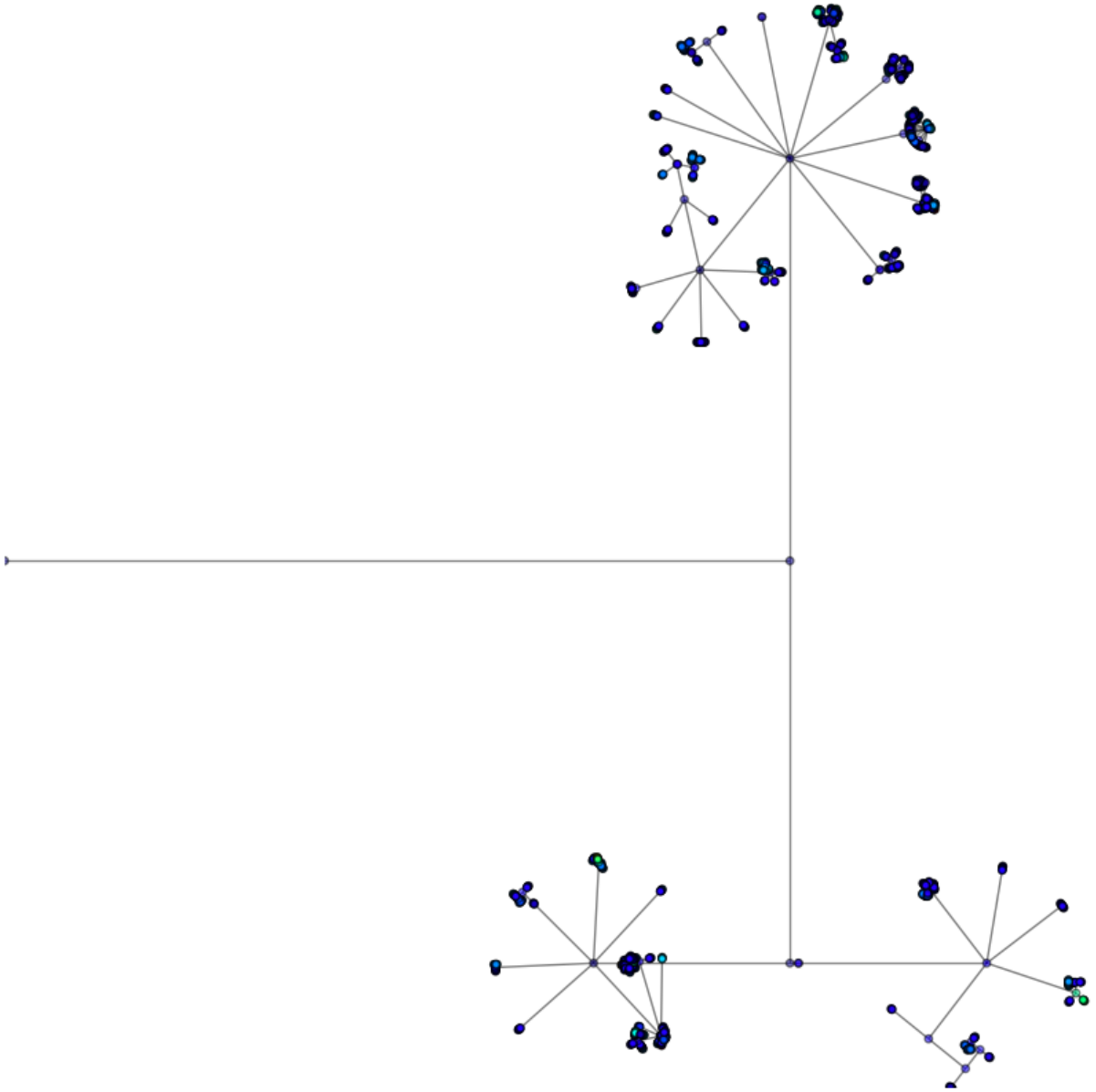
- Biconnected component (or block or 2-connected component) is a maximal biconnected subgraph
- Any connected graph decomposes into a tree of biconnected components (block cut tree) of the graph
- The blocks are attached to each at shared vertices called cut vertices or articulation points
- Cut vertex is any vertex whose removal increases the number of connected components



Each color corresponds to a biconnected component

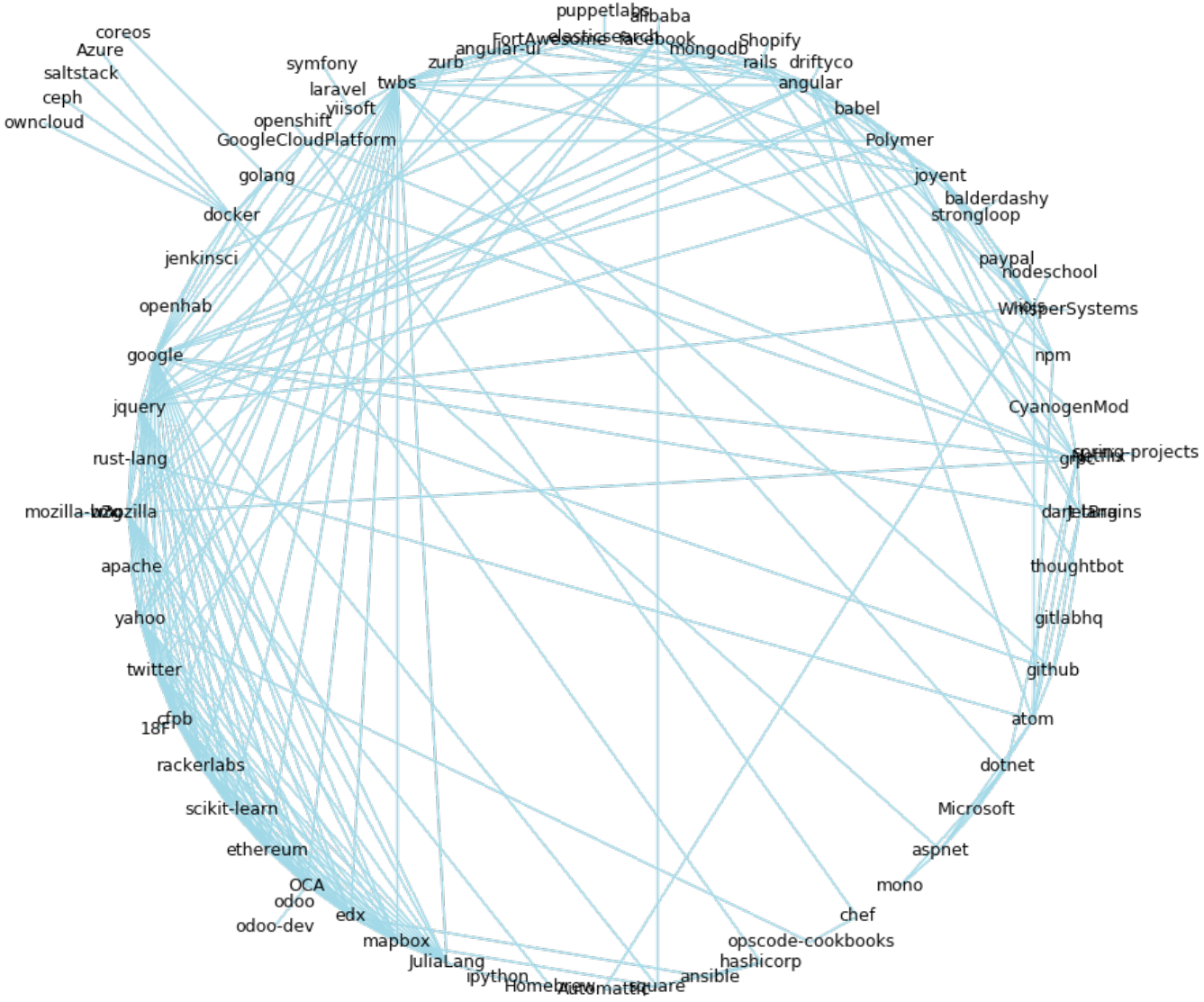
Graph layout: circo

- LANL routes example



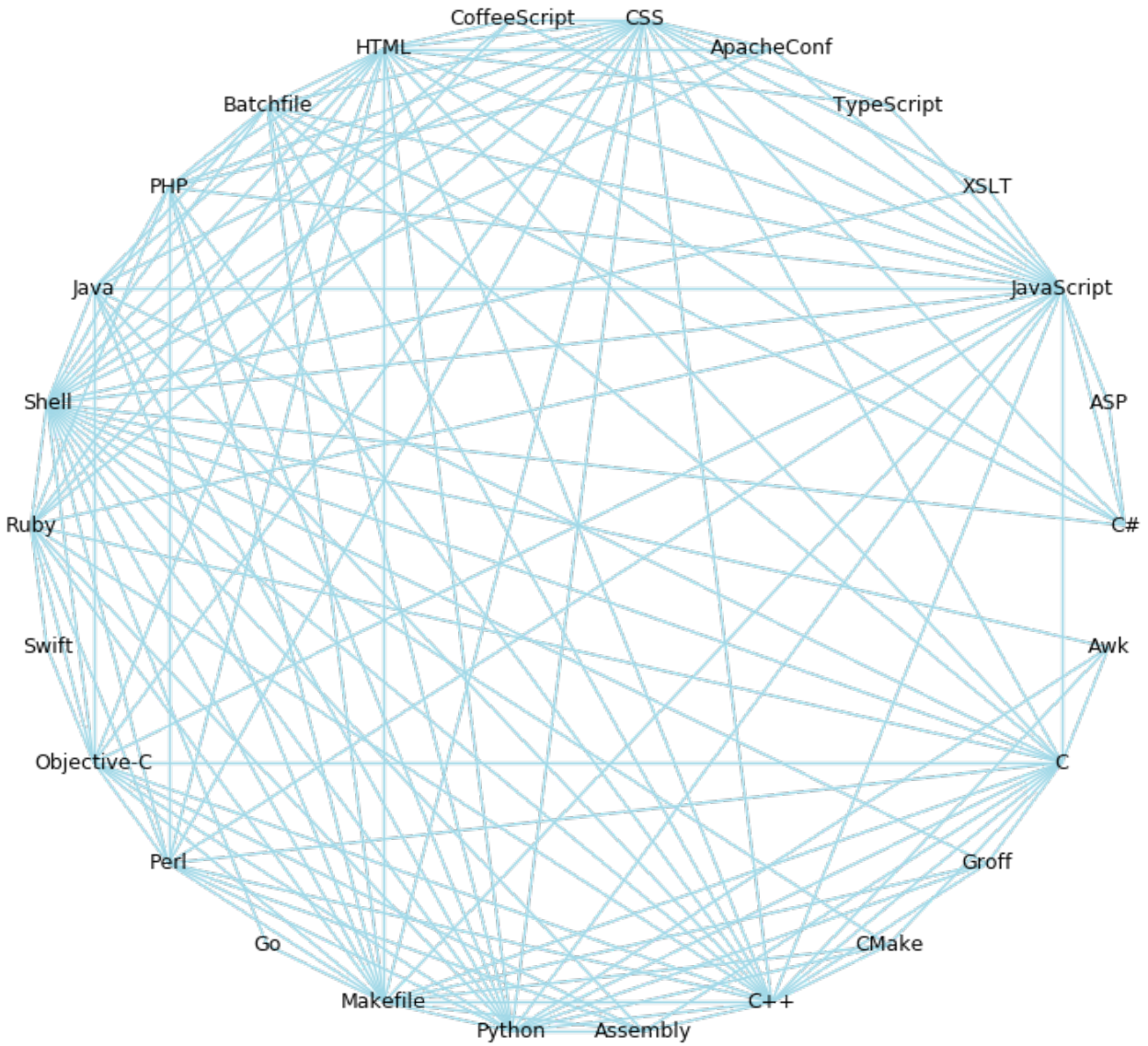
Graph layout: circo

- GithHub organizations example



Graph layout: circo

- GitHub programming languages example



Spectral layout

- Graph layout using eigenvectors of the graph Laplacian matrix

- It may produce a graph layout that positions groups of vertices very close to each other

```
def _spectral(A, dim=2):
    # Input adjacency matrix A
    # Uses dense eigenvalue solver from numpy
    try:
        import numpy as np
    except ImportError:
        raise ImportError("spectral_layout() requires numpy: http://scipy.org/ ")
    try:
        nnodes,_=A.shape
    except AttributeError:
        raise nx.NetworkXError(\
            "spectral() takes an adjacency matrix as input")

    # form Laplacian matrix
    # make sure we have an array instead of a matrix
    A=np.asarray(A)
    I=np.identity(nnodes,dtype=A.dtype)
    D=I*np.sum(A,axis=1) # diagonal of degrees
    L=D-A

    eigenvalues,eigenvectors=np.linalg.eig(L)
    # sort and keep smallest nonzero
    index=np.argsort(eigenvalues)[1:dim+1] # 0 index is zero eigenvalue
    return np.real(eigenvectors[:,index])
```

Force directed algorithms

- Also referred to as spring embedders
- Calculate the layout of a graph only using information contained in the structure of the graph, not on relying on any domain-specific knowledge
- Graphs drawn with these algorithms tend to be aesthetically pleasing, exhibit symmetries and produce crossing-free layouts for planar graphs
- Basic idea: define an objective function (energy function) which maps each graph layout into a positive real number representing the energy of the layout
- The energy function is defined in such a way that low energies correspond to layouts in which adjacent nodes are near some pre-specified distance from each other and non-adjacent nodes are well-spaced

Force directed algorithms (cont'd)

- The goal is to position nodes of a graph in a two-dimensional or a three-dimensional space such that all the edges are approximately of equal length and there are as few crossing edges as possible

- This is attempted by assigning forces among the nodes and edges, based on their relative positions, and then using these forces to stimulate the motion of the nodes to minimise an energy function

- Two types of forces:
  - Attractive force: based on Hooke's law used to attract nodes connected by edges
  - Repulsive force: like that those of electrically charged particles based on the Coulomb's law used to separate all pairs of nodes

History timeline

- 1963 Tutte: force-directed methods for drawing a graph
  - Polyhedral graphs may be drawn in a plane with all faces convex by fixing the vertices of the outer face of a planar embedding of the graph into convex position, placing a spring-like attractive force on each edge, and letting the system settle into an equilibrium
  - This system cannot get stuck at a local minima
  - Tutte's embedding – embedding of planar graphs with convex faces
- 1984 Eades: combination of attractive forces between adjacent vertices and repulsive forces between all vertices
- 1989 Kamada and Kawai: use of spring forces proportional to graph distances
- 1991 Fruchterman and Reingold: similar algorithm as that of Eades

Tutte’s barycentric algorithm

- An algorithm introduced in the seminal work by Tutte (1963)

- Basic idea: if a face of the planar graph is fixed in the plane, then suitable positions for the remaining vertices can be found by solving a system of linear equations, where each vertex position is represented as a convex combination of the positions of its neighbors

Tutte's barycentric algorithm (cont'd)

- Each vertex $v \in V$ is associated with a point $p_v \in \mathbf{R}^2$ and a force

$$f_v(p) = \sum_{(u,v)\in E} \|p_v - p_u\|^2$$

- Objective function: $f(p) = \sum_{v\in V} f_v(p)$

- Using the notation $p_v = (x_v, y_v)$ for $v \in V$ we can write

$$f(p) = x^T L_A x + y^T L_A y$$

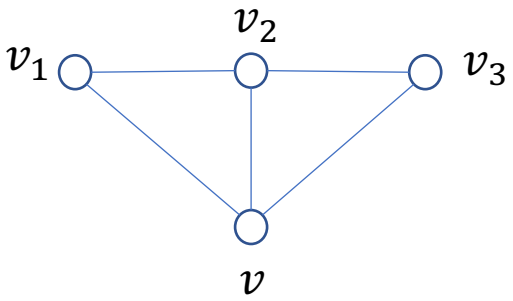
Stationary point

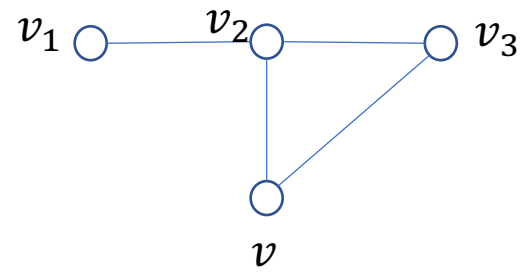
- Stationary point: (x, y) such that $L_A x = 0$ and $L_A y = 0$
- Equivalent to

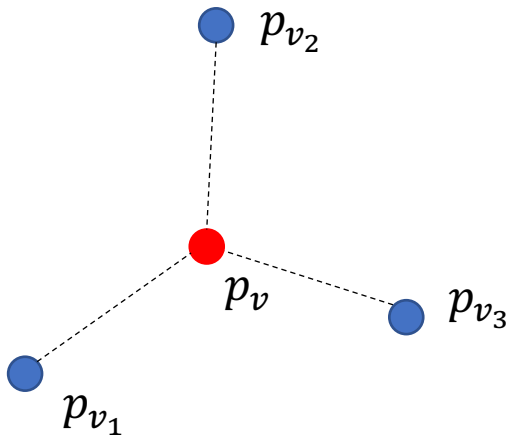
$$x = D_A^{-1} A x \quad \text{and} \quad y = D_A^{-1} A y$$

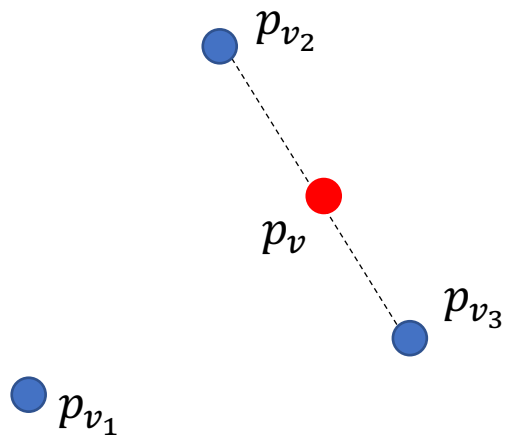
- This is a system of linear equations that has a unique solution
- Solution of these equations places its each free vertex at the barycenter of its neighbors

Barycenteric points









Pseudo code

Input: $G = (V, E)$, partition $V = V_0 \cup V_1$ such that V_0 is a set of at least three *fixed* vertices and V_1 is a set of *free* vertices; strictly convex polygon P with $|V_0|$ vertices

Output: positions p_v for $v \in V$ such that the fixed vertices form a convex polygon P

Init: place each fixed vertex $v \in V_0$ at a vertex of P and each free vertex at the origin

repeat

foreach free vertex $v \in V_1$ **do**

$$x_v = \frac{1}{\deg(v)} \sum_{(u,v) \in E} x_u$$

$$y_v = \frac{1}{\deg(v)} \sum_{(u,v) \in E} y_u$$

until x_v and y_v converge for all free vertices v

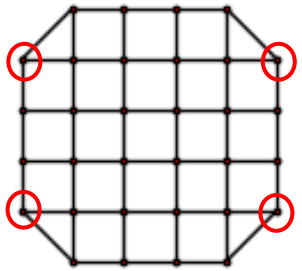
Tutte's theorem (1963)

Definitions:

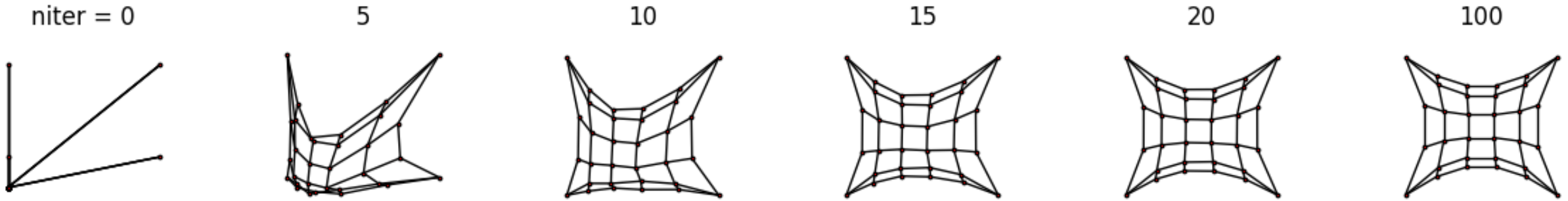
- A graph is 3-connected if it cannot be made disconnected by removing two vertices
- A graph is planar if it can be drawn in the plane without crossing edges
- A drawing of a planar graph such that the drawing has no crossing edges divides the plane into connected regions called faces

Theorem: Let G be a 3-connected planar graph and let B be the set of vertices on a face of G. If we fix the positions of the vertices in B so that the edges between them enclose a strictly convex polygon in $\mathbf{R}^2$ and set every other vertex to the average of its neighbors, then the resulting embedding of G is planar.

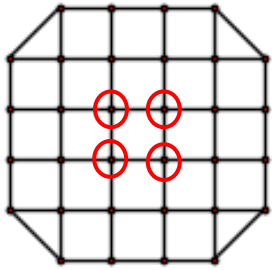
Example: racket



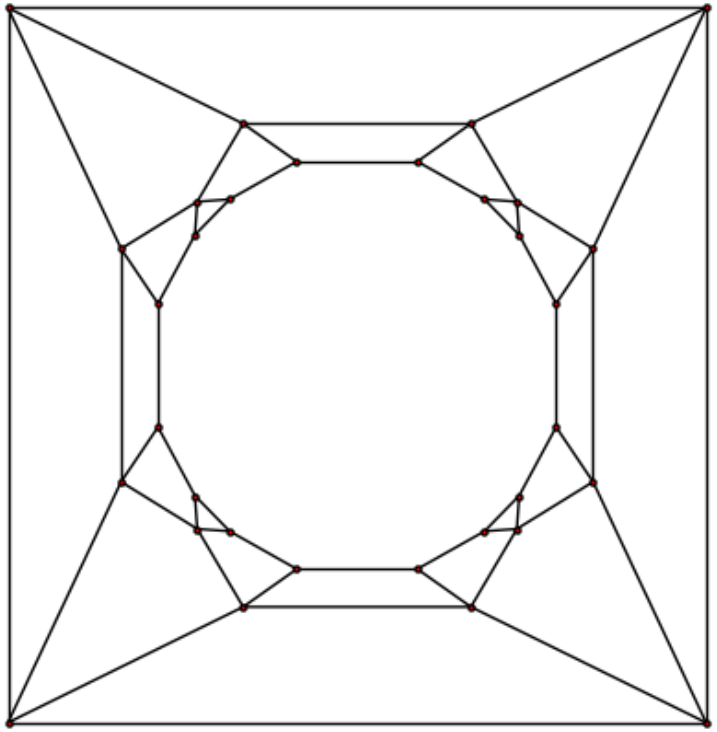
- Fixed vertices = (0,1), (0,4), (5,1), (5,4)
- Positions of fixed vertices = (0,1), (0,4), (5,1), (5,4)



Example: racket



niter = 100



- Fixed vertices = (2,2), (2,3), (3,2), (3,3)
- Positions of fixed vertices = (2,2), (2,3), (3,2), (3,3)

Some drawbacks of Tutte's algorithm

- Resulting drawing may have a poor vertex resolution

- For every number of two or more vertices there exists a graph such that the barycenter method computes a drawing with exponential area

  (Eades and Garvan, 1995)

Fruchterman and Reingold's algorithm

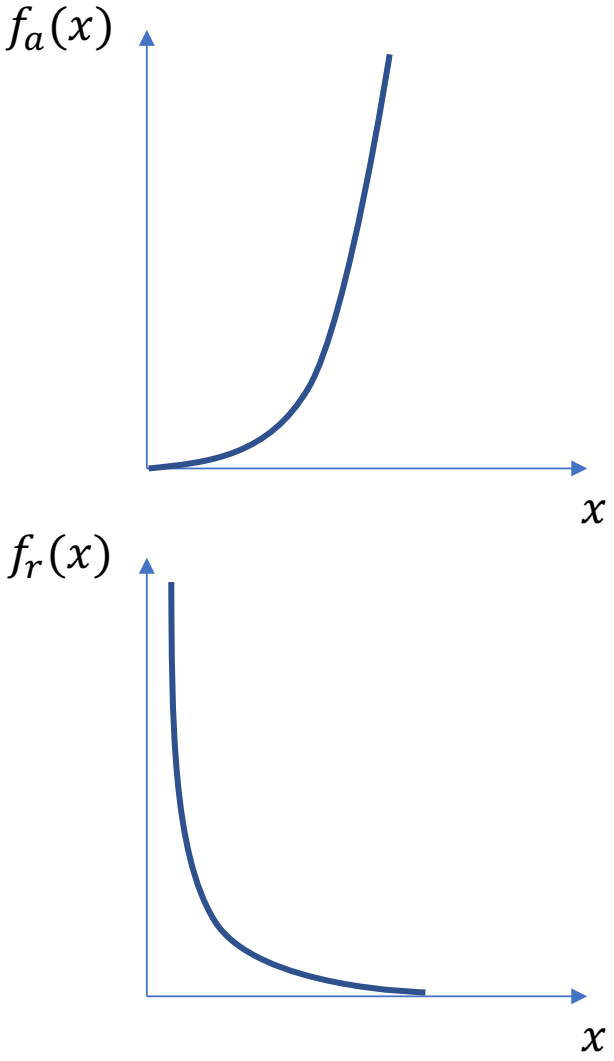
- L := height of a rectangular graph drawing area
- W := width of a rectangular graph drawing area

- Parameter $k = \sqrt{LW/|V|}$

- Two types of force functions:

  Attractive force: $f_a(x) = x^2/k$

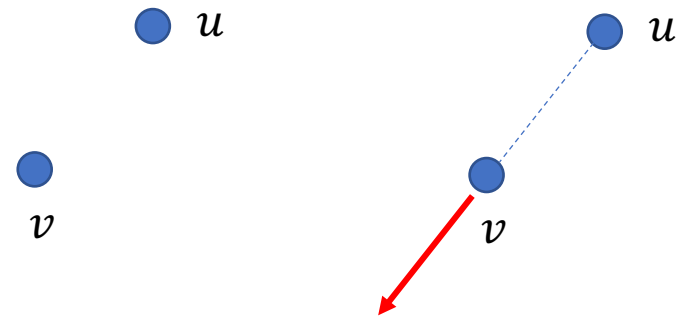
  Repulsive force: $f_r(x) = k^2/x$



Repulsive forces

Each vertex v associated with vectors $v.pos$ and $v.disp$

for $v \in V$:
    $v.disp \leftarrow 0$
    for $u \in V$:
        if $u \neq v$:
            $\delta \leftarrow v.pos - u.pos$
            $v.disp \leftarrow v.disp + \frac{\delta}{|\delta|} f_r(|\delta|)$



Attractive forces

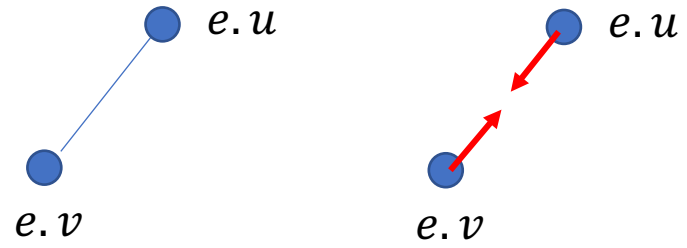
Each edge e is an ordered pair of vertices denoted as $(e.v, e.u)$

for $e \in E$:

$$\delta \leftarrow e.v.pos - e.u.pos$$

$$e.v.disp \leftarrow e.v.disp - \frac{\delta}{|\delta|} f_a(|\delta|)$$

$$e.u.disp \leftarrow e.u.disp + \frac{\delta}{|\delta|} f_a(|\delta|)$$



Limiting maximum displacement

For $v \in V$:

$$v.pos \leftarrow v.pos + \frac{v.disp}{|v.disp|}\min(v.disp, t)$$

$$v.pos.x \leftarrow \min\left(\frac{W}{2}, \max\left(v.pos.x, -\frac{W}{2}\right)\right)$$

$$v.pos.y \leftarrow \min\left(\frac{L}{2}, \max\left(v.pos.y, -\frac{L}{2}\right)\right)$$

Reduce temperature parameter t

Code

```
def _fruchterman_reingold(A, dim=2, k=None, pos=None, fixed=None,
                          iterations=50):
    # Position nodes in adjacency matrix A using Fruchterman-Reingold
    # Entry point for NetworkX graph is fruchterman_reingold_layout()
    try:
        import numpy as np
    except ImportError:
        raise ImportError("_fruchterman_reingold() requires numpy: http://scipy.or

    try:
        nnodes,_=A.shape
    except AttributeError:
        raise nx.NetworkXError(
            "fruchterman_reingold() takes an adjacency matrix as input")

    A=np.asarray(A) # make sure we have an array instead of a matrix

    if pos==None:
        # random initial positions
        pos=np.asarray(np.random.random((nnodes,dim)),dtype=A.dtype)
    else:
        # make sure positions are of same type as matrix
        pos=pos.astype(A.dtype)
```

```
    # optimal distance between nodes
    if k is None:
        k=np.sqrt(1.0/nnodes)
    # the initial "temperature"  is about .1 of domain area (=1x1)
    # this is the largest step allowed in the dynamics.
    # We need to calculate this in case our fixed positions force our domain
    # to be much bigger than 1x1
    t = max(max(pos.T[0]) - min(pos.T[0]), max(pos.T[1]) - min(pos.T[1]))*0.1
    # simple cooling scheme.
    # linearly step down by dt on each iteration so last iteration is size dt.
    dt=t/float(iterations+1)
    delta = np.zeros((pos.shape[0],pos.shape[0],pos.shape[1]),dtype=A.dtype)
    # the inscrutable (but fast) version
    # this is still O(V^2)
    # could use multilevel methods to speed this up significantly
```

Code (cont'd)

```
for iteration in range(iterations):
    # matrix of difference between points
    for i in range(pos.shape[1]):
        delta[:,:,i]= pos[:,i,None]-pos[:,i]
    # distance between points
    distance=np.sqrt((delta**2).sum(axis=-1))
    # enforce minimum distance of 0.01
    distance=np.where(distance<0.01,0.01,distance)
    # displacement "force"
    displacement=np.transpose(np.transpose(delta)*\
                              (k*k/distance**2-A*distance/k))\
                              .sum(axis=1)
    # update positions
    length=np.sqrt((displacement**2).sum(axis=1))
    length=np.where(length<0.01,0.1,length)
    delta_pos=np.transpose(np.transpose(displacement)*t/length)
    if fixed is not None:
        # don't change positions of fixed nodes
        delta_pos[fixed]=0.0
    pos+=delta_pos
    # cool temperature
    t-=dt
return pos
```

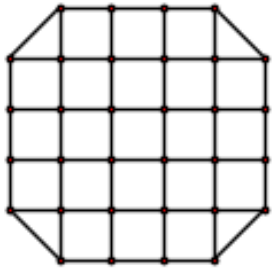
Graph layout: FR

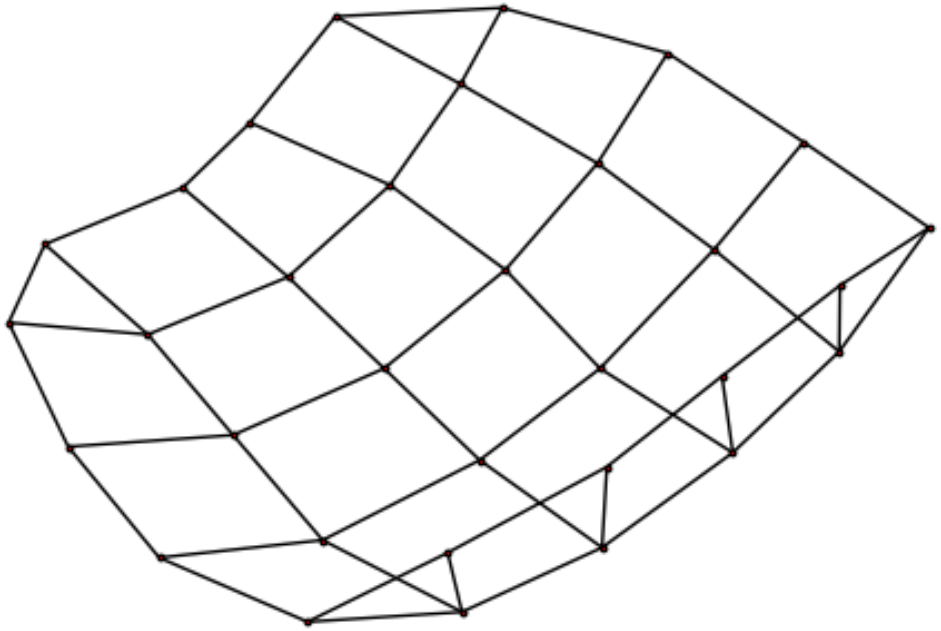
- GitHub organizations example
- NetworkX: spring layout
- Graphviz: fdp

Graph layout: FR vs number of iterations

FR for planar 3-connected graphs

- FR does not guarantee no crossing edges for planar 3-connected graphs





Algorithms using graph theoretic distances

- Approach proposed by Kamanda and Kawai (1989): "*We regard the desirable geometric (Euclidean) distance between two vertices in the drawing as the graph theoretic distance between them in the corresponding graph.*"

- Key idea: a spring system such that minimizing the energy of the system corresponds to minimizing the difference between the Euclidean and graph distances

- There are no separate attractive and repulsive forces between pairs of vertices

- Instead, if a pair of vertices is closer or farther away according to Euclidean distance than their corresponding graph distance the vertices repel or attract each other

KK algorithm: energy function

- Let $d_{u,v}$ be the shortest path distance between vertices u and v
- Define $\ell_{u,v} = C\ d_{u,v}$ to be the desirable length of a single edge in the graph drawing, for a positive-valued parameter C
- The energy function is defined as:

$$F(p) = \frac{1}{2}\sum_{u<v} \frac{K}{d_{u,v}^2}\left(|p_u - p_v| - \ell_{u,v}\right)^2$$

- A stationary point can be found using the Newton-Raphson method

Pseudo code

while $\max_v \left\| \nabla_{p_v} F(p) \right\| > \epsilon$:

Let $v^* \in \arg\max_v \left\| \nabla_{p_v} F(p) \right\|$

while $\left\| \nabla_{p_{v^*}} F(p) \right\| > \epsilon$:

Find $\delta x, \delta y$ that is a solution to: $[\nabla^2_{p_v} F(p) + \nabla_{p_v} F(p)] \begin{pmatrix} \delta x \\ \delta y \end{pmatrix} = 0$

$p_{v^*} \leftarrow p_{v^*} + \begin{pmatrix} \delta x \\ \delta y \end{pmatrix}$

Relation with multidimensional scaling

- The energy function corresponds to the metric-based multidimensional scaling by replacing $K/d_{u,v}^2$ with 1 and replacing $\ell_{u,v}$ with $d_{u,v}$ (recall week 10 lecture)

- The multidimensional scaling objective is referred to as stress minimization

- The stress minimization can be globally minimized via a majorization iterative method

Limitations of force directed algorithms

- The use of basic force-directed algorithms is limited to small graphs and results are poor for graphs with more than a few hundred vertices
  - One reason is that the objective function has many poor local minima

- Techniques to alleviate these shortcomings:
  - Multi-level layout: graph represented by a series of progressively simpler structures, laid out in reverse order from the simplest to the most complex

- Classical force-directed algorithms are restricted to computing graph layouts in Euclidean spaces (for visualizations, 2 or 3 dimensional)
  - Some graphs better represented in non-Euclidean spaces (ex. sphere or torus)

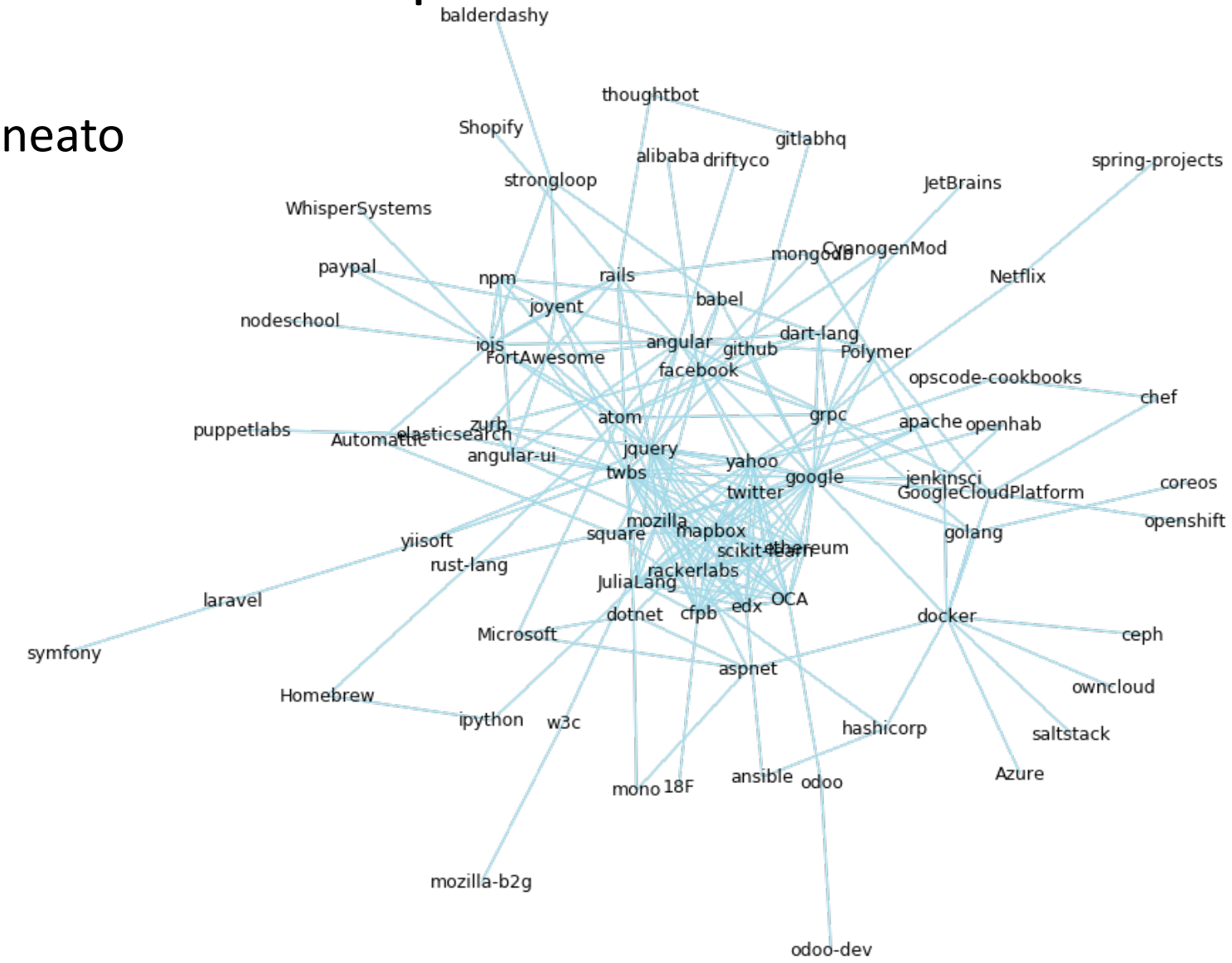
Computation complexity

- General running time $O(n^3)$
  - Linear number of iterations with quadratic complexity per iteration
  - High-dimensional embedding

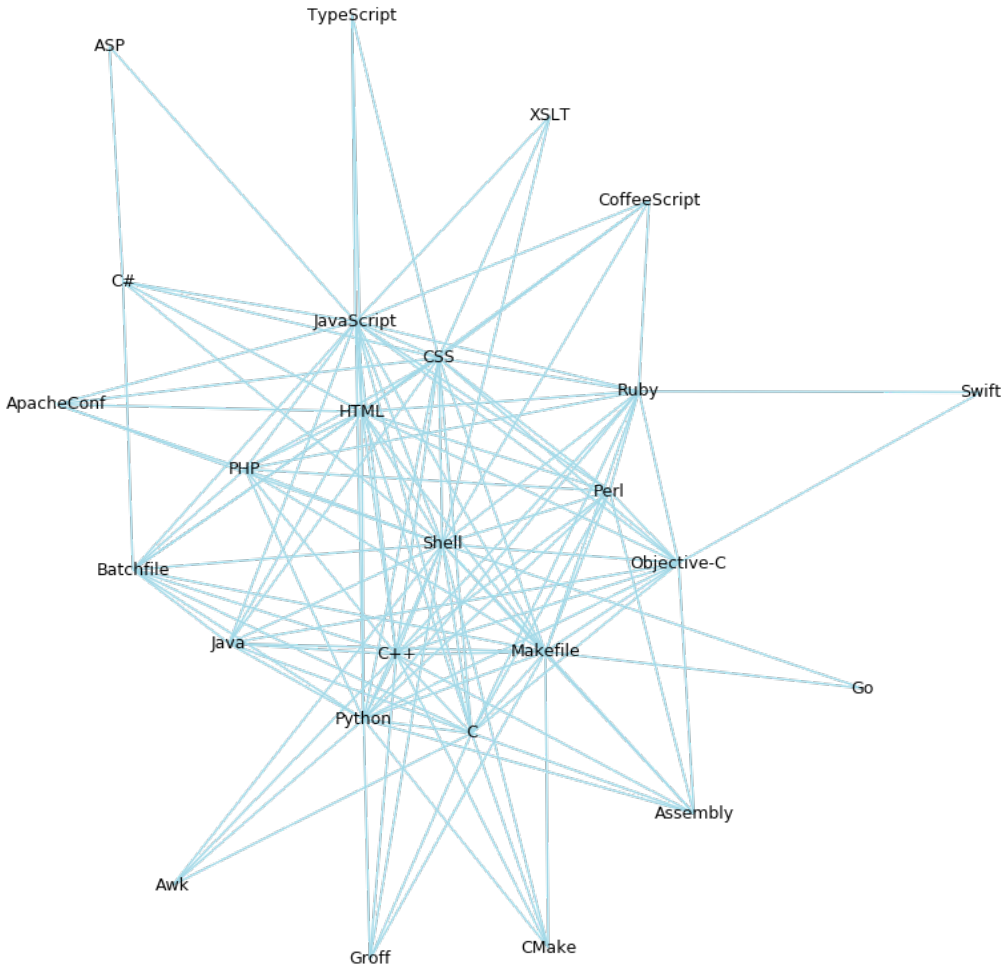
- Barnes-Hut simulation-based method
  - Running time $O(n \log n)$ per iteration
  - Results in running time $O(n^2 \log n)$

Graph layout: GitHub repositories

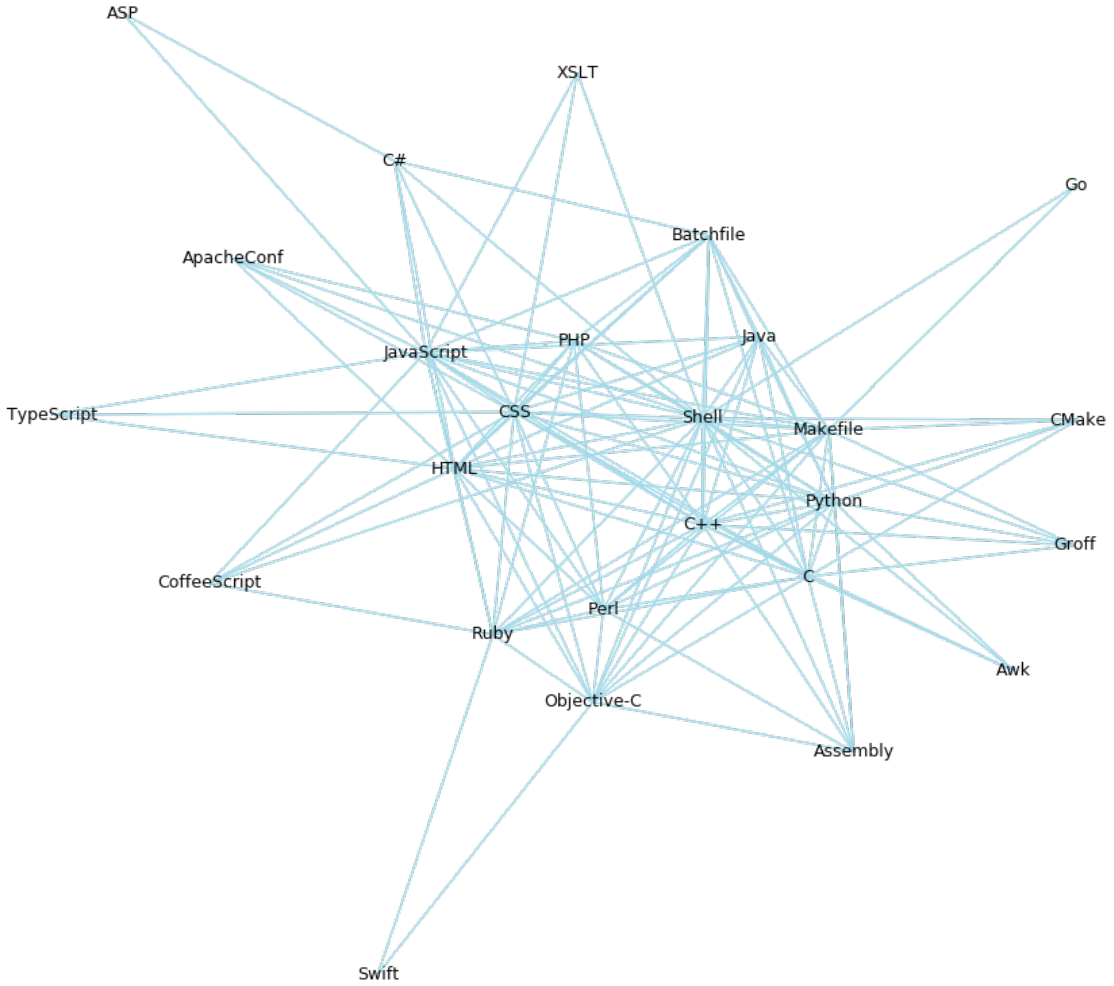
- Graphviz graph layout neato



Graph layouts: GitHub programming languages



neato



fdp

Summary

- No single algorithm works best for all input graph instances
- Graph visualization algorithms are based on principles such as spring embedding, spectral embedding, and radial layouts
- Spring embedding methods have cubic or quadratic running time complexity in the number of input graph vertices, limiting their scalability

References

- G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis, Graph Drawing: Algorithms for the Visualization of Graphs, Prentice Hall, Englewood Cliffs, NJ, 1999
- T. M. J. Fruchterman and E. M. Reingold, Graph drawing by force-directed placement, Software-practice and experience, 21 (11), 1991, pp 1129-1164
- Kamada and Kawai, An algorithm for drawing general undirected graphs, Information Processing Letters, Vol 31, No 1, 1989, pp 7-15
- S. G. Koburov, Force-directed drawing algorithms, Chapter 12, R. Tamassia (Editor), Handbook of graph drawing and visualization, pp. 383-408, CRC Press, 2013
- Y. Koren, Drawing graphs by eigenvectors: Theory and practice, Computers & Mathematics with Applications, 49, pp 1867-1888, 2005

References

- J. M. Six and I. G. Tollis, Circular drawings of biconnected graphs, Algorithm Engineering and Experimentation: International Workshop ALENEX'99, Selected Papers, Lecture Notes in Computer Science, 1999
- R. Tamassia (Editor), Handbook of graph drawing and visualization, CRC Press, 2013 http://cs.brown.edu/~rt/gdhandbook/
- W. T. Tutte, How to draw a graph, Proc. London Math. Soc. (3) 13, 1963, pp 743-68
- G. J. Wills, NicheWorks-interactive visualization of very large graphs, Proc. of Graph Drawing, 1997

Further references

- NetworkX Reference https://media.readthedocs.org/pdf/networkx/latest/networkx.pdf
- E. R. Gansner, Using Graphviz as a library (cgraph version), Graphviz library, manual, 2014
- S. C. North, Neato Guide, 2004 http://www.graphviz.org/pdf/neatoguide.pdf
- Phylogenetic tree viewer - http://www.aaronvose.net/phytree3d/
- Graphdrawing.org http://graphdrawing.org/
- Graph drawing https://en.wikipedia.org/wiki/Graph_drawing
- Gephi - https://gephi.org/
- Wolfram Graph drawing introduction - http://reference.wolfram.com/language/tutorial/GraphDrawingIntroduction.html