Homework Assignment 2: Network Flows

5273

Circular Layout

As the name suggest this layout positions nodes in a circle [13]. According to [14] although this algorithm can appear trivial, it is widely used to visualize complexes and pathways. The algorithm attempts to minimize the number of overlapping nodes and edges, this way interactions among nodes are easier to understand and locate. It allows to highlight the node or nodes with the highest degree, moreover it evidences potencially interesting sub-networks as inner circles in the network [1].

```
import networkx as nx
import matplotlib.pyplot as plt

G=nx.Graph()
G.add_nodes_from(['A','B','C','D','E'])

G.add_edges_from([('A','B'),('B','C'),('C','D'),('D','E'),('E','A')])

nx.draw(G,pos=nx.circular_layout(G),with_labels=True)  #Draw the Graph named G with the circular layout
plt.savefig("Graph02.eps", format="EPS")
plt.show()
```

graph02ucg.py

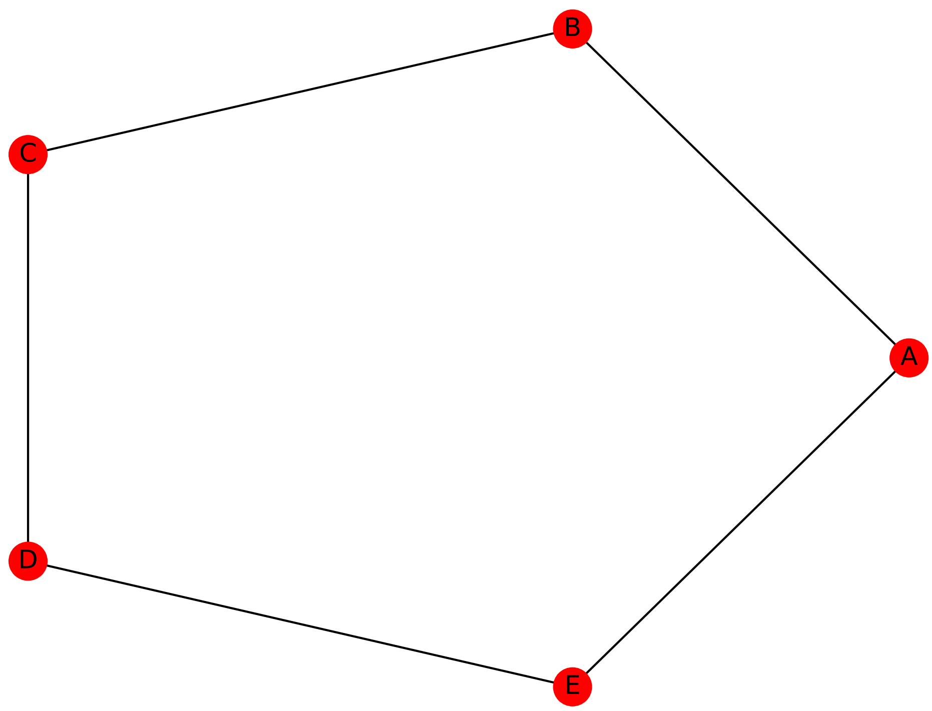

Figure 1: Circular layout

Random Layout

This algorithm positions nodes by choosing coordinates uniformly at random on the interval [0.0, 1.0) where the nodes are generated [13]. The advantage of this kind of layout lays in the very fast drawing on the network graph. However it carries the disadvantage of the difficulty in grasping the position of nodes [12].

```python
import networkx as nx
import matplotlib.pyplot as plt

R=nx.DiGraph()
R.add_nodes_from(["D","1","2","3","4"])

R.add_edges_from([("D","1"),("1","2"),("2","D")])
R.add_edges_from([("D","3"),("3","4"),("4","D"),("D","D")])

color_map = []
for node in R:
    if (node == "D"):
        color_map.append('blue')
    else:
        color_map.append('red')

nx.draw(R, node_color=color_map, pos=nx.random_layout(R), with_labels=True)
plt.savefig("Graph06.eps", format="EPS")
plt.show()
```

graph06drg.py

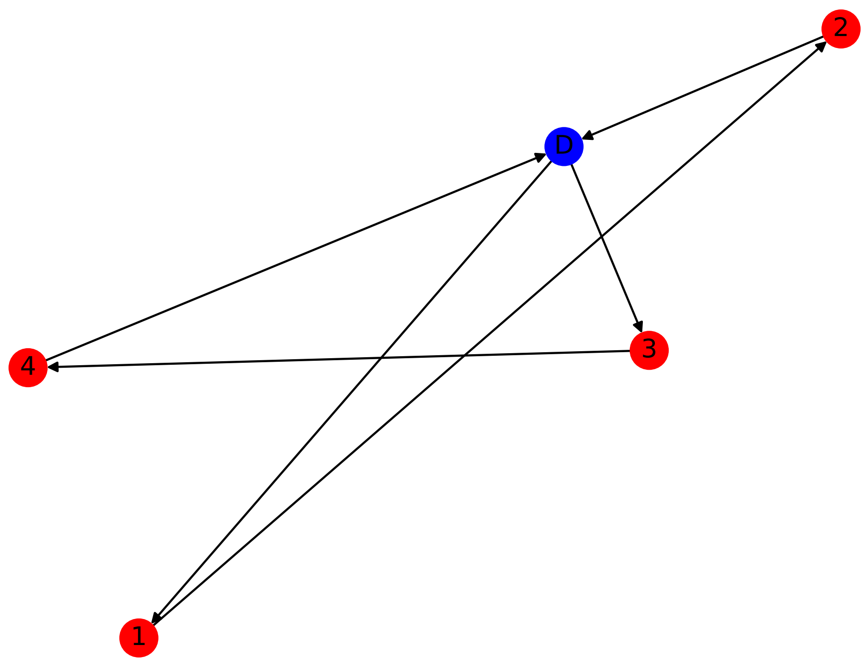

Figure 2: Random layout

Bipartite Layout

This layout positions nodes in two straight lines [13]. This particular class only allows the connection between two nodes in different sets [4]. Nodes from set X are only connected with nodes from set Y, not with other nodes from X, and vice versa [9].

```python
import networkx as nx
import matplotlib.pyplot as plt

M=nx.MultiDiGraph()
M.add_nodes_from(["Coahuila","Veracruz"], bipartite=0)
M.add_nodes_from(["Sonora","Guanajuato","Quintana_Roo"], bipartite=1)

M.add_edge("Sonora","Coahuila", color='blue', weight=1)
M.add_edges_from([("Coahuila","Sonora"),("Coahuila","Guanajuato"),("Guanajuato","Veracruz"),("Veracruz","Quintana_Roo"),("Quintana_Roo","Coahuila")],color='black', weight=1)

edges = M.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(M.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

nx.draw(M, pos=nx.bipartite_layout(M,["Coahuila","Veracruz"]),edges=edges,edge_color=colors,
    width=weight,with_labels=True)
plt.savefig("Graph11.eps", format="EPS")
plt.show()
```

graph11dcm.py

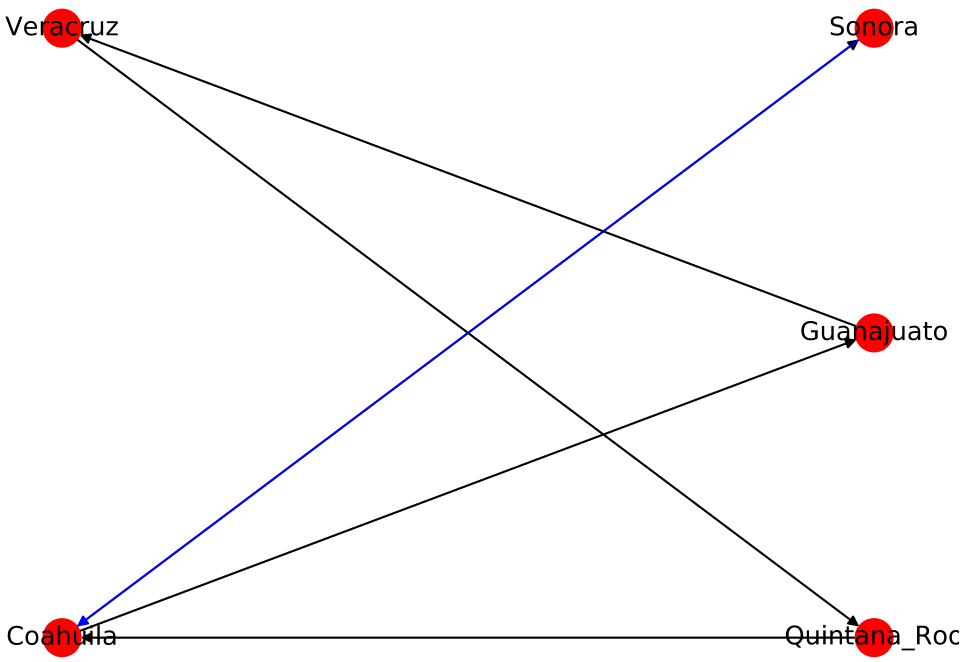

Figure 3: Bipartite layout

Kamada-Kawai Layout

Is an algorithm for drawing undirected graphs and weighted graphs and is based on the concept of theoretic distance between nodes [8]. In this algorithm the forces between the nodes can be determined by the lengths of shortest paths between each couple of nodes [13]. The classical Kamada-Kawai algorithm does not scale well when it is used in networks with large numbers of nodes [2].
This layout will be represented by an undirected reflexive graph and a directed acyclic multigraph.

Undirected reflexive graph

```
import networkx as nx
import matplotlib.pyplot as plt

H=nx.Graph()
H.add_nodes_from([0,1,2])
H.add_nodes_from([3,4])

H.add_edges_from([(0,1),(0,2),(1,2),(0,3),(0,4),(3,3)])

color_map = []
for node in H:
    if (node == 3):
        color_map.append('blue')
    else:
        color_map.append('red')

pos = nx.kamada_kawai_layout(H)

nx.draw(H, pos=pos, node_color=color_map, with_labels=True)
plt.savefig("Graph03.eps", format="EPS")
plt.show()
```

graph03urg.py

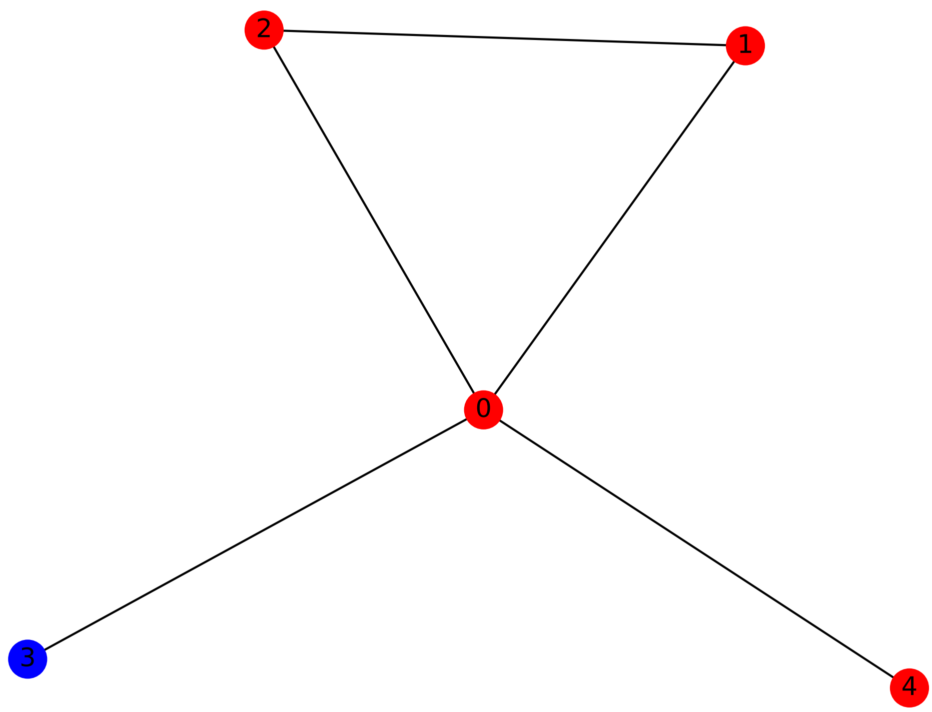

Figure 4: Kamada-Kawai layout for the undirected reflexive graph

Directed acyclic multigraph

```python
import networkx as nx
import matplotlib.pyplot as plt

M=nx.MultiDiGraph()                          #Create an empty directed multigraph

M.add_nodes_from(["x1","x2","x3","x4","x5"])

M.add_edge("x1","x2",color='green',weight=6)
M.add_edges_from([("x1","x2"),("x3","x2"),("x4","x3"),("x4","x2"),("x5","x2")],color='black',
    weight=1)

edges = M.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(M.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

pos=nx.kamada_kawai_layout(M)

nx.draw(M,pos=pos,edges=edges,edge_color=colors,width=weight,with_labels=True)
plt.savefig("Graph10.eps", format="EPS")
plt.show()
```

graph10dam.py

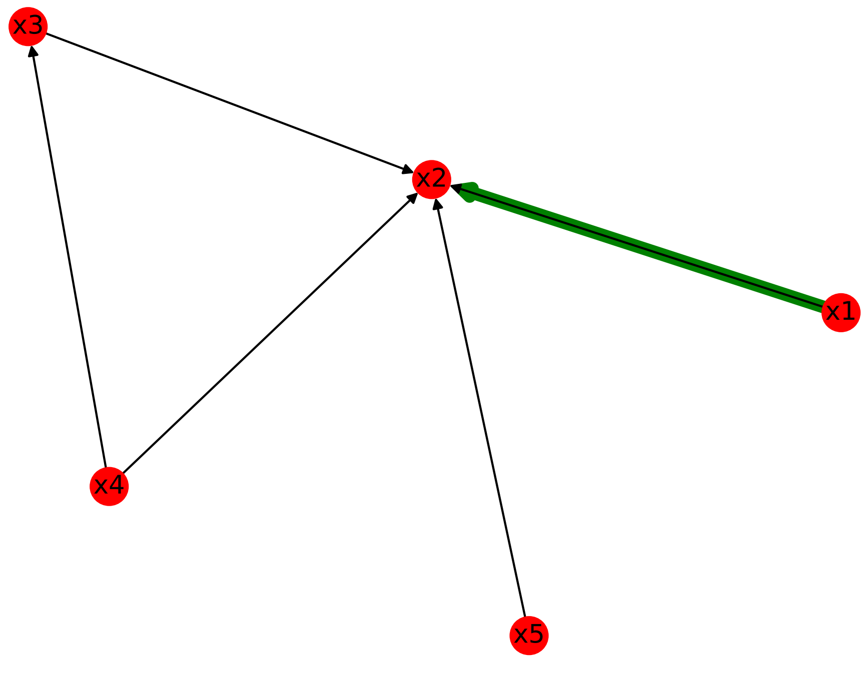

Figure 5: Kamada-Kawai layout for the directed acyclic multigraph

Shell Layout

This layout positions nodes in concentric circles [13]. It is also known as concentric layout in [3], where it is said that this arrangement in a series of concentric circles is based on the distance from the central node. Thus, nodes with direct connections are arranged in the first circle, then the nodes located at a distance of two nodes away from the center follows in this arrengement and so on, until the graph is completed. The main advantage of this layout is that viewers are able to see network structures easier and to navigate them noticing the closeness of relationships to a single node to each other.
This layout will be represented by a directed cyclic graph and a directed reflexive multigraph.

Directed cyclic graph

```
import networkx as nx
import matplotlib.pyplot as plt

G = nx.DiGraph()
G.add_node('DEPOT')
G.add_nodes_from(['Job1','Job2','Job3','Job6'])
G.add_nodes_from(['Job4','Job5','Job7'])
G.add_nodes_from(['Job8','Job9'])

G.add_path(['DEPOT','Job1','Job2','Job3','Job6','DEPOT'])  #Construct a path from the nodes in that order
G.add_path(['DEPOT','Job4','Job5','Job7','DEPOT'])
G.add_path(['DEPOT','Job8','Job9','DEPOT'])

pos=nx.shell_layout(G)
nx.draw(G,pos=pos,with_labels=True)
plt.savefig("Graph05.eps", format="EPS")
plt.show()
```

graph05dcg.py

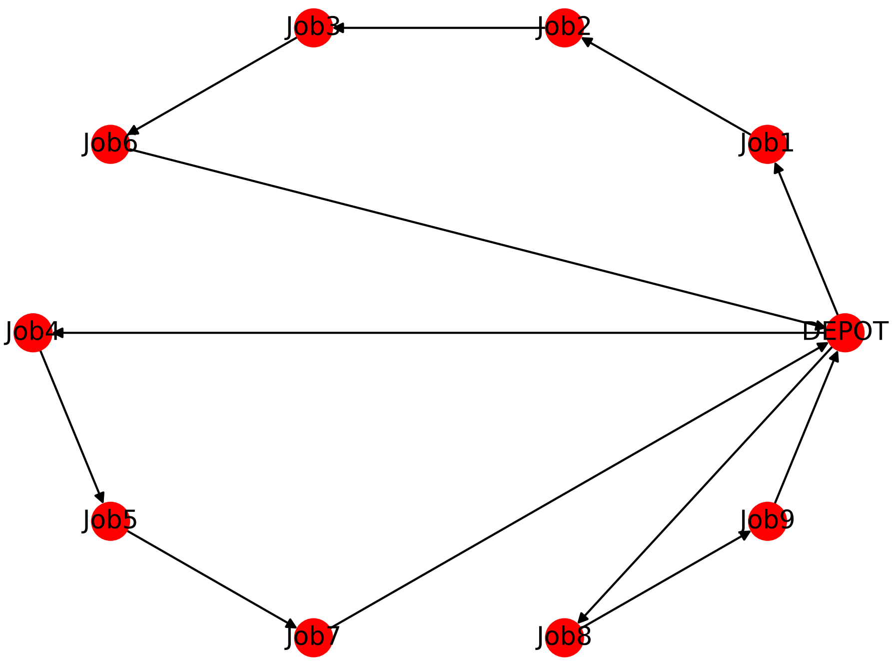

Figure 6: Shell layout for the directed cyclic graph

Directed reflexive multigraph

```
import networkx as nx
import matplotlib.pyplot as plt

M=nx.MultiDiGraph()

M.add_nodes_from(["Depot","Warehouse1","Warehouse2","Warehouse3","Warehouse4"])

M.add_edges_from([("Depot","Warehouse1"),("Depot","Warehouse1"),("Depot","Warehouse2"),("Depot","Warehouse3"),("Depot","Warehouse4"),("Depot","Depot")])

color_map = []
for node in M:
    if (node == 3):
        color_map.append('blue')
    else:
        color_map.append('red')

pos=nx.shell_layout(M)

nx.draw(M, pos=pos, node_color=color_map, with_labels=True)
plt.savefig("Graph12.eps", format="EPS")
plt.show()
```

graph12drm.py

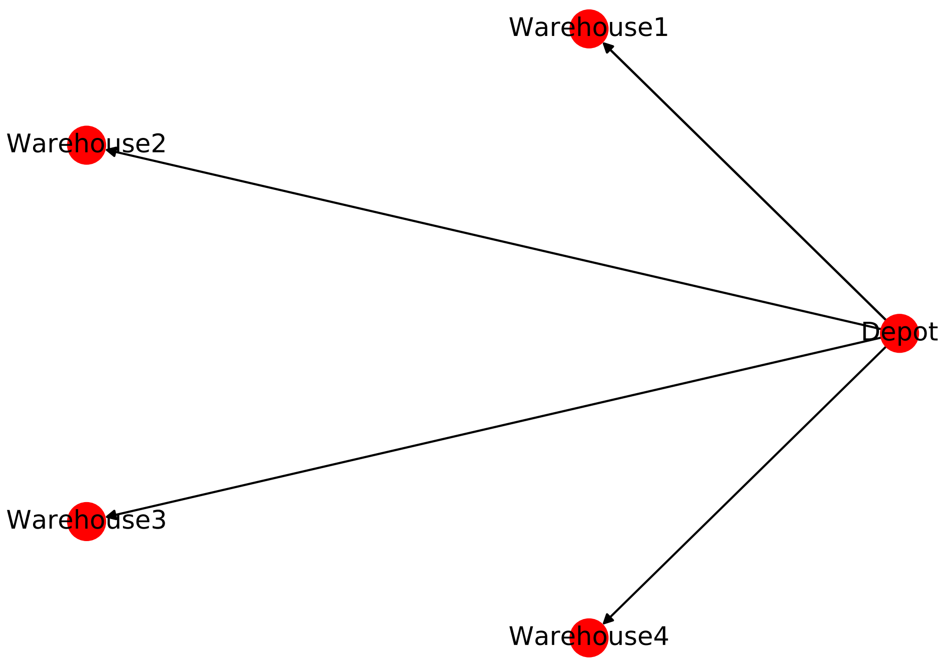

Figure 7: Shell layout for the directed cyclic graph

Fruchterman-Reingold Layout

This algorithm attempts to distribute vertices evenly, make edge lengths uniform, and reflect symmetry [5]. Is best for fewer than thirty nodes and does not require parameters to be optimised [6]. This layout will be represented by an undirected acyclic multigraph and a directed acyclic graph.

Undirected acyclic multigraph

```
import networkx as nx
import matplotlib.pyplot as plt

M = nx.MultiGraph()                                    #Create an empty Multigraph

M.add_nodes_from(['x','y','z','v','w'])

M.add_edge('x','y',color='blue',weight=6)
M.add_edges_from([('x','y'),('x','y'),('y','z'),('z','v'),('v','w')],color='black',weight=1)

edges = M.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(M.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

pos = nx.fruchterman_reingold_layout(M,k=0.15,iterations=20)

nx.draw(M, pos=pos, edges=edges, edge_color=colors, width=weight, with_labels=True)
plt.savefig("Graph07.eps", format="EPS")
plt.show()
```

graph07uam.py

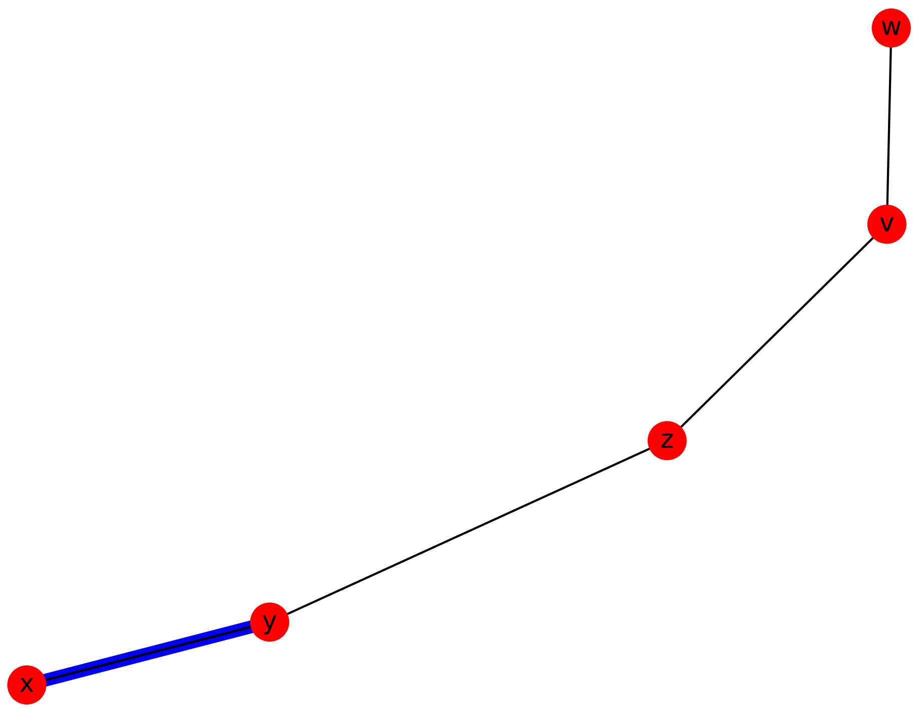

Figure 8: Fruchterman-Reingold layout for the undirected acyclic multigraph

Directed acyclic graph

```python
import networkx as nx
import matplotlib.pyplot as plt

G=nx.DiGraph()                                    #Create an empty directed graph
G.add_node("X")
G.add_nodes_from(["R1","R2","R3","R4"])

G.add_edges_from([("X","R1"),("X","R2"),("X","R3"),("X","R4")])

pos=nx.fruchterman_reingold_layout(G,k=0.2,iterations=30)

nx.draw(G,pos=pos,node_color='skyblue',with_labels=True)
plt.savefig("Graph04.eps", format="EPS")
plt.show()
```

graph04dag.py

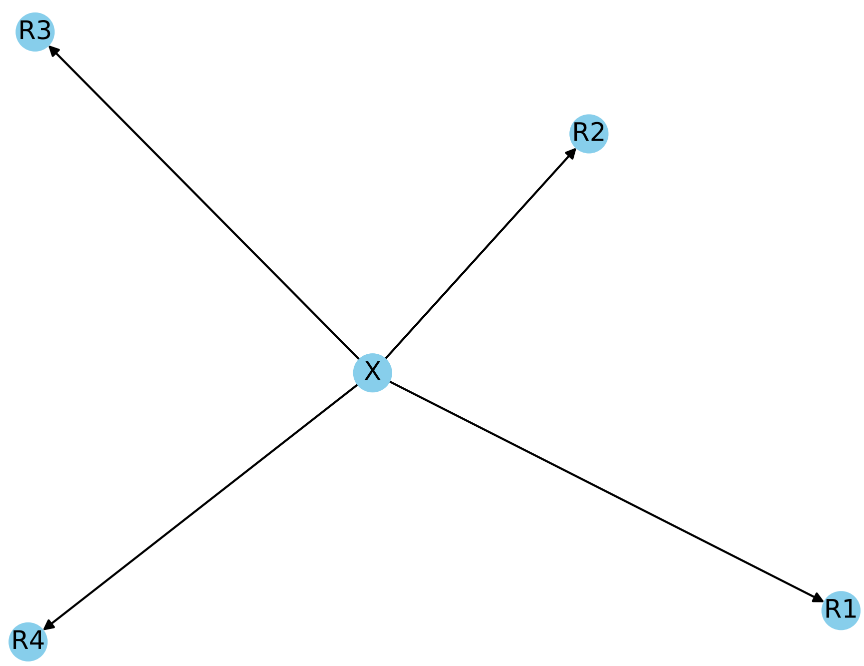

Figure 9: Fruchterman-Reingold layout for the directed acyclic graph

Spring Layout

This layout positions nodes using the algorithm of Fruchterman-Reingold [13]. The algorithm first places the vertices in some initial layout and then it moves the rings in which the system reach a minimal energy due to the spring forces. With this layout we should obtain a display as much symmetry as possible [10].

```python
import networkx as nx
import matplotlib.pyplot as plt

M=nx.MultiGraph()

M.add_nodes_from([1,2,3,4,5])

M.add_edge(2,1, color='blue',weight=6)
M.add_edges_from([(1,2),(1,3),(3,4),(4,5),(5,1)],color='black', weight=1)

edges = M.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(M.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

pos = nx.spring_layout(M, scale=3)

nx.draw(M,pos,edges=edges,edge_color=colors,width=weight,with_labels=True, font_size=8,
    font_family='sans-serif')
plt.savefig("Graph08.eps", format="EPS")
plt.show()
```

graph08ucm.py

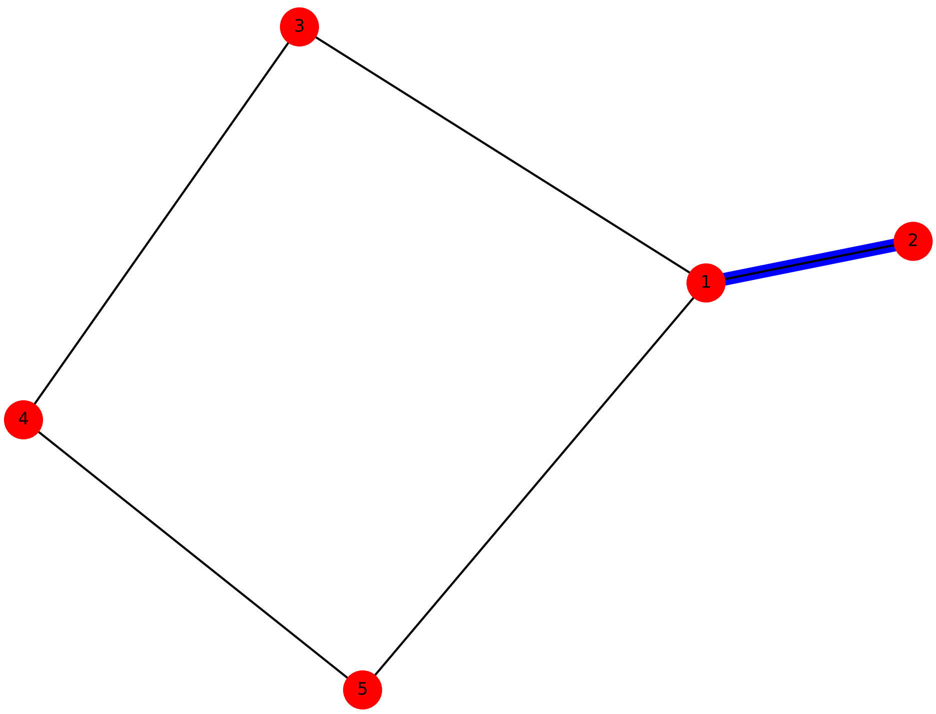

Figure 10: Spring layout

Spectral Layout

This approach has the advantage of computing optimal layouts (according to some requirements) and rapid computation time. It provides an exact solution to the layout problem, whereas other formulations end in an NP-hard problem with approximated solutions [11]. To construct the layout it uses eigenvectors of the adjacency matrix of the graph and of the Laplacian spectrum associated with the graph [11]. While we use this layout in Python using NetworkX, positioning the nodes, directed graphs will be considered as unidrected graphs [13].

```
import networkx as nx
import matplotlib.pyplot as plt

M=nx.MultiGraph()
M.add_nodes_from(["A","B","C","D","E"])

M.add_edges_from([("A","B"),("A","C")],color='blue', weight=8)
M.add_edges_from([("A","A"),("A","B"),("A","C"),("B","C"),("B","D"),("C","E"),("D","E")],
    color='black', weight=2)

edges = M.edges()

colors = []
weight = []

for (u,v,attrib_dict) in list(M.edges.data()):
    colors.append(attrib_dict['color'])
    weight.append(attrib_dict['weight'])

g=nx.shell_layout(M)
nx.draw(M, edges=edges, pos=g, edge_color=colors, width=weight, with_labels=True)
#plt.savefig("Graph09.eps", format="EPS")
plt.show()
```

graph09urm.py

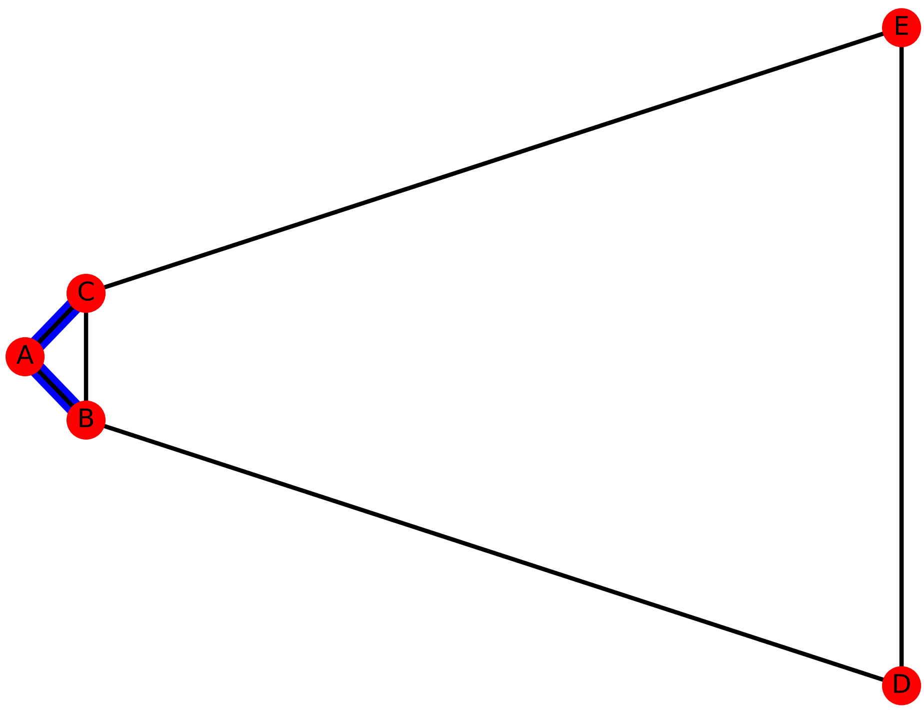

Figure 11: Spring layout

Force Atlas 2 Layout

Force Atlas 2 is a very fast layout algorithm for force-directed graphs. It's used to spatialize a weighted undirected graph in 2D (Edge weight defines the strength of the connection). The implementation is based on [7]. Its really quick compared to the fruchterman reingold algorithm (spring layout) of networkx and scales well to high number of nodes.

```python
import networkx as nx
from fa2 import ForceAtlas2
import matplotlib.pyplot as plt

G = nx.Graph()                                          #Create an empty graph

G.add_node('Libro')                                     #Add a simple node
G.add_nodes_from(['C1','C2','C3'])                      #Add a list of nodes
G.add_nodes_from(['s1.1','s1.2','s2.1','s2.2','s2.3'])
G.add_nodes_from(['s2.1.1','s2.1.2'])

G.add_edges_from([('Libro','C1',),('Libro','C2',),('Libro','C3')])
G.add_edges_from([('C1','s1.1',),('C1','s1.2',)])
G.add_edges_from([('C2','s2.1',),('C2','s2.2',),('C2','s2.3')])
G.add_edges_from([('s2.1','s2.1.1'),('s2.1','s2.1.2')])

#Consulted at: |https://github.com/bhargavchippada/forceatlas2
forceatlas2 = ForceAtlas2(
                        # Behavior alternatives
                        outboundAttractionDistribution=True,  # Dissuade hubs
                        linLogMode=False,
                        adjustSizes=False,
                        edgeWeightInfluence=1.0,

                        # Performance
                        jitterTolerance=1.0,
                        barnesHutOptimize=True,
                        barnesHutTheta=1.2,
                        multiThreaded=False,

                        # Tuning
                        scalingRatio=2.0,
                        strongGravityMode=False,
                        gravity=1.0,

                        # Log
                        verbose=True)

positions = forceatlas2.forceatlas2_networkx_layout(G, pos=None, iterations=2000)

nx.draw_networkx_nodes(G, positions, node_size=100, with_labels=True, node_color="red", alpha=0.4)
nx.draw_networkx_edges(G, positions, edge_color="black", alpha=0.05)
plt.axis('off')
plt.savefig("Graph01.eps", format="EPS")
plt.show()
```

graph01uag.py

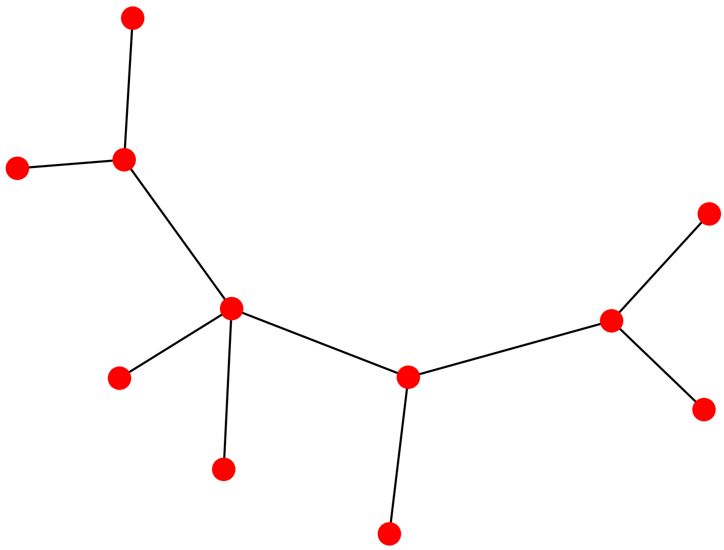

Figure 12: Force Atlas 2 layout

References

[1] Giuseppe Agapito, Pietro Hiram Guzzi, and Mario Cannataro. Visualization of protein interaction networks: problems and solutions. *BMC bioinformatics*, 14(1):3–6, 2013.

[2] Se-Hang Cheong and Yain-Whar Si. Accelerating the kamada-kawai algorithm for boundary detection in a mobile ad hoc network. *CoRR*, abs/1508.05312, 2015. URL `http://arxiv.org/abs/1508.05312`.

[3] Ken Cherven. *Mastering Gephi network visualization*. Packt Publishing Ltd, 2015.

[4] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world*. Cambridge University Press, 2010.

[5] Thomas Fruchterman and Edward Reingold. Graph drawing by force-directed placement. *Software: Practice and experience*, 21(11):1129–1164, 1991.

[6] Helen Gibson, Joe Faith, and Paul Vickers. A survey of two-dimensional graph layout techniques for information visualisation. *Information visualization*, 12(3-4):324–357, 2013.

[7] Mathieu Jacomy, Tommaso Venturini, Sebastien Heymann, and Mathieu Bastian. Forceatlas2, a continuous graph layout algorithm for handy network visualization designed for the gephi software. *PloS one*, 9(6): e98679, 2014.

[8] Tomihisa Kamada, Satoru Kawai, et al. An algorithm for drawing general undirected graphs. *Information processing letters*, 31(1):7–15, 1989.

[9] Stefan Kasberger. Introduction into bipartite networks with python, February 10 2014. Karl Franzens University of Graz, Peter Csermely.

[10] Stephen G. Kobourov. Spring embedders and force directed graph drawing algorithms. *CoRR*, abs/1201.3011, 2012. URL `http://arxiv.org/abs/1201.3011`.

[11] Yehuda Koren. On spectral graph drawing. In *International Computing and Combinatorics Conference*, pages 496–508. Springer, 2003.

[12] K. Mouri, H. Ogata, N. Uosaki, and M. Kiyota. Visualization for analyzing learning logs in the seamless learning environment. In *Proceedings of the 24th International Conference on Computers in Education. India: Asia-Pacific Society for Computers in Education*, pages 315–324, 2016.

[13] NetworkX. Source code for networkx.drawing.layout, 2018. `https://networkx.github.io/documentation/latest/_modules/networkx/drawing/layout.html`, Last accessed on 2019-02-13.

[14] Jyh-Jong Tsay, Bo-Liang Wu, and Yu-Sen Jeng. Hierarchically organized layout for visualization of biochemical pathways. *Artificial intelligence in medicine*, 48(2-3):107–117, 2010.